



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações
e de Computadores



GUITAR SOLO TRACKER

Relatório Final

Alaney Kilson Dória
(Licenciado)

TRABALHO DE PROJECTO REALIZADO PARA OBTENÇÃO DO GRAU
DE MESTRE EM ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Júri:

Fernando Manuel Gomes de Sousa, Presidente do Júri
José Manuel Peixoto Nascimento, Vogal (arguente)
Manuel Martins Barata, Vogal (arguente)
Artur Jorge Ferreira, Vogal (orientador)

Março de 2011



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações
e de Computadores



GUITAR SOLO TRACKER

Relatório Final

Alaney Kilson Dória
(Licenciado)

TRABALHO DE PROJECTO REALIZADO PARA OBTENÇÃO DO GRAU
DE MESTRE EM ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Júri:

Fernando Manuel Gomes de Sousa, Presidente do Júri
José Manuel Peixoto Nascimento, Vogal (arguente)
Manuel Martins Barata, Vogal (arguente)
Artur Jorge Ferreira, Vogal (orientador)

Março de 2011

Resumo

A motivação para este trabalho vem da necessidade que o autor tem em poder registar as notas tocadas na guitarra durante o processo de improviso. Quando o músico está a improvisar na guitarra, muitas vezes não se recorda das notas tocadas no momento, este trabalho trata o desenvolvimento de uma aplicação para guitarristas, que permita registar as notas tocadas na guitarra eléctrica ou clássica. O sinal é adquirido a partir da guitarra e processado com requisitos de tempo real na captura do sinal.

As notas produzidas pela guitarra eléctrica, ligada ao computador, são representadas no formato de tablatura e/ou partitura. Para este efeito a aplicação capta o sinal proveniente da guitarra eléctrica a partir da placa de som do computador e utiliza algoritmos de detecção de frequência e algoritmos de estimação de duração de cada sinal para construir o registo das notas tocadas.

A aplicação é desenvolvida numa perspectiva multi-plataforma, podendo ser executada em diferentes sistemas operativos Windows e Linux, usando ferramentas e bibliotecas de domínio público.

Os resultados obtidos mostram a possibilidade de afinar a guitarra com valores de erro na ordem de 2 Hz em relação às frequências de afinação standard. A escrita da tablatura apresenta resultados satisfatórios, mas que podem ser melhorados. Para tal será necessário melhorar a implementação de técnicas de processamento do sinal bem como a comunicação entre processos para resolver os problemas encontrados nos testes efectuados.

Palavras chave: Reconhecimento automático de notas musicais, Transcrição musical, Detecção e estimação de frequências, Fast Fourier Transform, Algoritmo de Goertzel, Auto-correlação.

Abstract

The motivation for this work comes from the author's need to be able to record the notes played on guitar during the process of improvisation. When the musician is improvising on the guitar, often does not remember the notes played at the time, this work addresses the development of an application for guitarists. The application aims to register the notes played on a classic or electric guitar. The signal is obtained from the guitar and processed with real-time requirements for audio capture.

The notes produced by the electric guitar, connected to the computer, are presented in the form of a tablature and/or score. In order to get this result, the application captures the signal of a guitar, by the sound card of the computer, and uses frequency estimation and detection algorithms and signal duration algorithms, in order to create the register of the notes played.

The application is developed in a multi-platform perspective, allowing to be run in different operating systems, using tools and public domain libraries.

The obtained results show a possibility of tuning the guitar with an error within a 2 Hz range in relation to the frequency tuning standard. However the writing of the tablature has not obtained conclusive results regarding the level of reliability of the application. Therefore improvements of the implementation of the signal processing in order to solve the problems found during testing and interprocess communication, will be necessary.

Keywords: Automatic recognition of musical notes, Musical transcription, Detection and estimations of frequencies, Fast Fourier Transform, Goertzel algorithm, Autocorrelation.

Índice

1	Introdução.....	1
1.1	Soluções existentes.....	3
1.2	Descrição geral da solução proposta.....	4
1.3	Organização do texto.....	5
2	Formulação do problema.....	7
2.1	Conceitos e terminologia associados à música.....	7
2.1.1	Notação musical.....	10
2.1.2	Representação das notas.....	11
2.2	Guitarra eléctrica e equipamentos.....	15
2.2.1	Produção de som da guitarra eléctrica.....	15
2.2.2	Escala e frequências da guitarra eléctrica.....	17
2.2.3	Hardware e software.....	18
2.3	Problema a resolver.....	19
3	Técnicas de processamento de sinal para detecção de frequência.....	20
3.1	Algoritmo FFT.....	20
3.2	Algoritmo Goertzel.....	21
3.3	Auto-correlação.....	23
3.4	Análise comparativa das soluções.....	24
3.5	Filtragem e decimação.....	26
3.5.1	Filtro passa-baixo de Chebyshev.....	31
3.5.1.1	Implementação do Filtro.....	32
3.6	Técnicas de janelas.....	33
3.7	Técnica de estimação da duração da nota.....	33
3.8	Resumo.....	36
4	Solução proposta e implementação.....	39
4.1	Crítérios de desenho da solução.....	39
4.2	Arquitectura.....	41
5	Aspectos de implementação e desenvolvimento.....	43
5.1	GST Transporter	43
5.1.1	Transporter comandos	46
5.1.2	Classe CLogger.....	47
5.1.3	Classe Frame.....	48
5.1.4	Classe Config e ConfigManager.....	48
5.1.5	Classe CTimer.....	49
5.1.6	Timer no sistema operativo Windows.....	50
5.1.7	Timer no sistema operativo Linux.....	50
5.2	Camada de Áudio.....	50
5.2.1	Classe CAudioServer.....	51
5.2.2	Classe CAudioPort.....	52
5.2.3	Classe CBuffer.....	53
5.3	Camada de processamento de sinal.....	54
5.3.1	Classe CSignalProcessing.....	55

5.4 Camada de gravação.....	56
5.4.1 Classe CAudioFileIO.....	56
5.5 Integração entre servidor e interface gráfica.....	57
5.5.1 Integração com ficheiro áudio.....	59
5.5.2 Integração com fila de mensagem.....	60
5.6 Interface gráfica.....	61
5.6.1 Classe Transporter.....	63
5.6.2 Classe libSndFile.....	64
5.6.3 Janelas da aplicação.....	65
5.7 Testes unitários.....	68
5.7.1 Testes unitários do servidor Áudio.....	68
5.7.2 Testes unitários do Transporter.....	72
5.7.3 Teste unitário da Interface gráfica.....	73
6 Avaliação Experimental.....	75
6.1 Execução do benchmark.....	75
6.2 Afinação da guitarra.....	79
6.3 Detecção e apresentação das notas.....	81
6.3.1 Testes efectuados com a guitarra clássica.....	81
6.3.2 Testes efectuados com guitarra eléctrica.....	82
7 Conclusões.....	83
8 Anexo A - Tabelas com resultados de detecção de frequência.....	85
9 Anexo B - Cálculos dos coeficientes.....	87
Referências.....	89

Índice de figuras

Figura 1: Screenshot do Guitar Pro (adaptado de [3]).....	2
Figura 2: Screenshot da aplicação Guitar Rig (adaptado de [5]).....	2
Figura 3: Sintetizador Roland VG-99 (Adaptado de [6]).....	3
Figura 4: Captadores para captar as notas da guitarra (adaptado de [8]).....	4
Figura 5: Esquema de funcionamento da aplicação.....	5
Figura 6: Teclado de piano, as teclas brancas são notas naturais e as pretas acidentadas.....	7
Figura 7: Exemplo de uma oitava no piano.....	8
Figura 8: Exemplo do braço da guitarra com todas as notas (adaptado de [11]).....	8
Figura 9: Metrónomo analógico (adaptado de [13]).....	9
Figura 10: Metrónomo TM-40 (adaptado de [14]).....	9
Figura 11: Elementos de uma pauta musical.....	10
Figura 12: Clave de Sol, Fá e Dó.....	11
Figura 13: Notação musical para piano (adaptado de [15]).....	11
Figura 14: Tabela com figuras musicais para representação de notas e pausas (Adaptado de [16]).....	12
Figura 15: Exemplos de aplicação das figuras em diferentes compassos (adaptado de [17]).....	12
Figura 16: Relação entre as figuras musicais (adaptado de [18]).....	13
Figura 17: Exemplo de tablatura para guitarra.....	13
Figura 18: Exemplo de tablatura com efeitos ou técnicas (adaptado de [22])	14
Figura 19: Exemplo de representação de Bend Up (adaptado de [23]).....	14

Figura 20: Exemplo da técnica Legato Slide (adaptado de [23]).....	14
Figura 21: Guitarra eléctrica (adaptado de [24]).....	15
Figura 22: Captador de guitarra(adaptado de [25]).....	16
Figura 23: Funcionamento dos captadores (adaptado de [25]).....	16
Figura 24: Esquema de um captador magnético (adaptado de [26]).....	17
Figura 25: Diagrama de bloco do filtro ordem II de Goertzel.....	21
Figura 26: Diagrama de decimação.....	26
Figura 27: Resposta em frequência do filtro tipo Bessel de ordem 4 (adaptado de [31]). O eixo x representa as frequências em fracções da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.....	27
Figura 28: Resposta em frequência do filtro de Butterworth de ordem 4 (adaptado de [31]). O eixo x representa as frequências em fracções da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.....	28
Figura 29: Resposta em frequência do filtro de Chebyshev de ordem 4 (adaptado de [31]). O eixo x representa as frequências em fracções da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.....	28
Figura 30: Resposta em frequência do filtro de Bessel de ordem 8 (adaptado de [31]). O eixo x representa as frequências em fracções da frequência de amostragem, O eixo y a vermelho representa a magnitude linear e normalizada da resposta do filtro, e y a azul a fase.....	29
Figura 31: Resposta em frequência do filtro de Butterworth de ordem 8 (adaptado de [31]). O eixo x representa as frequências em fracções da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.....	30
Figura 32: Resposta em frequência do filtro de Chebyshev de ordem 8 (adaptado de [31]). O eixo x representa as frequências em fracções da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.....	30
Figura 33: Resposta em frequência do filtro de Chebyshev com ripple de 20% (adaptado de [33]).	31
Figura 34: Resposta em frequência do filtro de Chebyshev tipo I (adaptado de [34]).....	31
Figura 35: Resposta em frequência de filtro de Chebyshev tipo II (Adaptado de [34]).....	32
Figura 36: Diagrama de filtro IIR na forma directa II (Adaptado de [36]).....	32
Figura 37: Sequência da janela deslizante para o cálculo da duração do sinal.....	35
Figura 38: Exemplo de uma arquitectura multicore ou multi-processador (Adaptado de [44]).....	40
Figura 39: Comparação entre operações escalares e operações SIMD (adaptado de [46]).....	40
Figura 40: Arquitectura da aplicação.....	41
Figura 41: Camadas da aplicação.....	41
Figura 42: Diagrama de blocos resumindo as bibliotecas utilizadas e sua aplicação.....	42
Figura 43: Sequência de processamento do Transporter.....	44
Figura 44: Sequência para a estimação de frequências.....	45
Figura 45: Diagrama UML da classe CGSTTransporter.....	46
Figura 46: Diagrama UML da classe CLogger.....	47
Figura 47: Diagrama UML da classe Frame.....	48
Figura 48: Diagrama UML das classes Config e ConfigManager.....	48
Figura 49: Diagrama UML da camada de captura áudio.....	51
Figura 50: 1ª iteração entre o leitor e escritor.....	53
Figura 51: 2ª iteração entre leitor e escritor.....	53
Figura 52: M iteração entre leitor e escritor.....	54
Figura 53: Diagrama das classes da camada de processamento de sinal.....	55

Figura 54: Diagrama UML da classe CAudioFileIO.....	57
Figura 55: Diagrama de integração entre o GSTTransporter e a Interface gráfica.....	58
Figura 56: Diagrama geral da aplicação.....	63
Figura 57: Diagrama UML da classe GST.Transporter.....	64
Figura 58: Diagrama UML do namespace LibSndFile com classes e enumerados.....	65
Figura 59: Guitar Solo Tracker janela principal.....	66
Figura 60: Janela do Metrónomo.....	67
Figura 61: Janela do afinador de guitarra.....	67
Figura 62: Janela Transporter para iniciar e para o processo de detecção das notas.....	68
Figura 63: Interface gráfica do Jack Audio Connection Kit.....	69
Figura 64: Resultado da execução do teste.....	70
Figura 65: Picos máximos do sinal captado do microfone.....	70
Figura 66: Screenshoot do componente AudioTrack com desenho do sinal áudio em forma de onda	73
Figura 67: Erro de detecção de frequências (em Hz) com factor de decimação de 15, para as 49 no- tas.....	76
Figura 68: Erro de detecção de frequências com factor de decimação de 20, com as 49 notas.....	77
Figura 69: Erro de detecção de frequências com factor de decimação de 20, com as 45 notas descar- tando as notas com aliasing.....	78
Figura 70: Afinador Yamaha YT-120.....	80
Figura 71: Desenho da tablatura como resultado de notas tocadas na guitarra clássica.....	81
Figura 72: Braço da guitarra com as notas correspondentes a cada corda nos trastos (adaptado de [76]).....	84

Índice de tabelas

Tabela 1: As colunas representam as oitavas e as linhas a notas musicais.....	18
Tabela 2: Características dos computadores de desenvolvimento.....	18
Tabela 3: Número de pontos com base em Fs com resolução de 2 Hz.....	22
Tabela 4: Razão entre a frequência mínima 82,41 Hz e máxima 1174,66 Hz.....	25
Tabela 5: Complexidade dos algoritmos.....	25
Tabela 6: Parâmetros utilizados para analisar a resposta em frequência dos filtros passa-baixo.....	27
Tabela 7: Número de pontos da FFT, resolução no tempo e frequência.....	37
Tabela 8: Resumo dos parâmetros utilizados para estimação da frequência.....	37
Tabela 9: Parâmetros passados ao executável do Transporter	43
Tabela 10: Ficheiros e classes pertencentes ao GSTTransporter.....	46
Tabela 11: Comandos passados ao GSTTransporter.....	47
Tabela 12: Comparação das características das funções de temporizador disponibilizadas pela API do Windows.....	50
Tabela 13: Classes pertencentes à camada de áudio.....	51
Tabela 14: Classes pertencentes à camada de processamento áudio.....	54
Tabela 15: Classes e ficheiros pertencentes à camada de gravação áudio.....	56
Tabela 16: Classes pertencentes à interface gráfica.....	62
Tabela 17: Condições dos testes executados.....	68

Tabela 18: Lista de testes unitários executados no AudioServer.....	69
Tabela 19: Lista de testes unitários executados com CFileIO.....	70
Tabela 20: Lista de testes unitários executados com CSignalProcessing.....	71
Tabela 21: Lista de testes unitários executados com GSTTransporter.....	72
Tabela 22: Lista de testes unitários executados no GuitarSoloTrackerUI.....	73
Tabela 23: Frequências de cada corda na afinação standard.....	75
Tabela 24: Parâmetros e resultados de teste.....	75
Tabela 25: Resumo de resultados dos testes para encontrar o melhor factor de decimação para as 6 frequências.....	79
Tabela 26: Comparação do desvio do afinador da aplicação com uma guitarra clássica e guitarra eléctrica afinadas usando o afinador Yamaha YT-120.....	80
Tabela 27: Resultado de detecção de frequências com factor de decimação de 15. As frequências em amarelo correspondem às de afinação standard.....	85
Tabela 28: Resultado de detecção de frequências com factor de decimação de 20. As frequências em amarelo correspondem às de afinação standard.....	86

Lista de Acrónimos

ALSA – *Advance Linux Sound Architecture*

API – *Application Programming Interface*

ASIO – *Audio Stream Input Output*

BPM – *Beat Per Minute*

CPU – *Central Processing Unit*

DFT – *Discrete Fourier Transform*

DLL – *Dynamic Link Library*

FFT – *Fast Fourier Transform*

FFTW – *Fast Fourier Transform in the West*

FLAC – *Free Lossless Audio Codec*

GTK – *GIMP Tool Kit*

JACK – *Jack Audio Connection Kit*

LED – *Ligth Emmitting Diode*

MIDI – *Musical Instrument Digital Interface*

MSMQ – *Microsoft Message Queue*

PC – *Personal Computer*

SIMD – *Single Instruction Multiple Data*

SSE – *Streaming SIMD Extensions*

UML – *Unified Modeling Language*

XML – *Extensible Markup Language*

1 Introdução

A motivação para este trabalho vem da necessidade que o autor tem em poder registar as notas tocadas na guitarra durante o processo de improviso. Quando o músico está a improvisar na guitarra, muitas vezes não se recorda das notas tocadas no momento, este trabalho trata o desenvolvimento de uma aplicação para guitarristas, que permita registar as notas tocadas na guitarra eléctrica ou clássica. O sinal é adquirido a partir da guitarra e processado com requisitos de tempo real na captura do sinal.

No momento de inspiração de um músico, o improviso é normalmente o processo para novas criações. Para um guitarrista o processo normal para recordar o que foi improvisado, consiste em gravar o improviso e mais tarde ouvir e analisar a gravação, para posteriormente poder reproduzir e anotar o que foi tocado.

No caso de um guitarrista solo, existem técnicas como *leggato* [1], *shredding* [2], assim como *speed picking* [2] solo, em que são tocadas várias notas por batida; neste caso é difícil detectar todas as notas apenas com a audição do sinal à medida que este vai sendo produzido.

Designa-se de *speed picking* a capacidade de se tocar várias notas em BPM (Beat Per Minute) superiores a 160. A técnica de *sweeping* é uma forma de se conseguir os *speed picking*, tocando 6 cordas da guitarra, sequencialmente, para cima e para baixo, com a mão direita, como se estivesse a varrer as cordas, enquanto a mão esquerda pressiona a corda entre os trastos e toca 1 ou 2 notas por corda.

Para guitarristas, existem aplicações para criar e editar tablaturas como o Guitar Pro 6 [3], para fazer toda a composição na aplicação e ouvir em formato Musical Instrument Digital Interface (MIDI) [4]. O Guitar Pro permite também ver acordes, escalas e imprimir em formato de tablatura e partitura tudo o que o guitarrista compor. Contudo, esta aplicação não permite captar sinais da guitarra; o guitarrista tem que gravar a música que improvisou e ouvir para tentar reproduzir as notas tocadas. A Figura 1 apresenta o *screenshot* do software Guitar Pro 6.

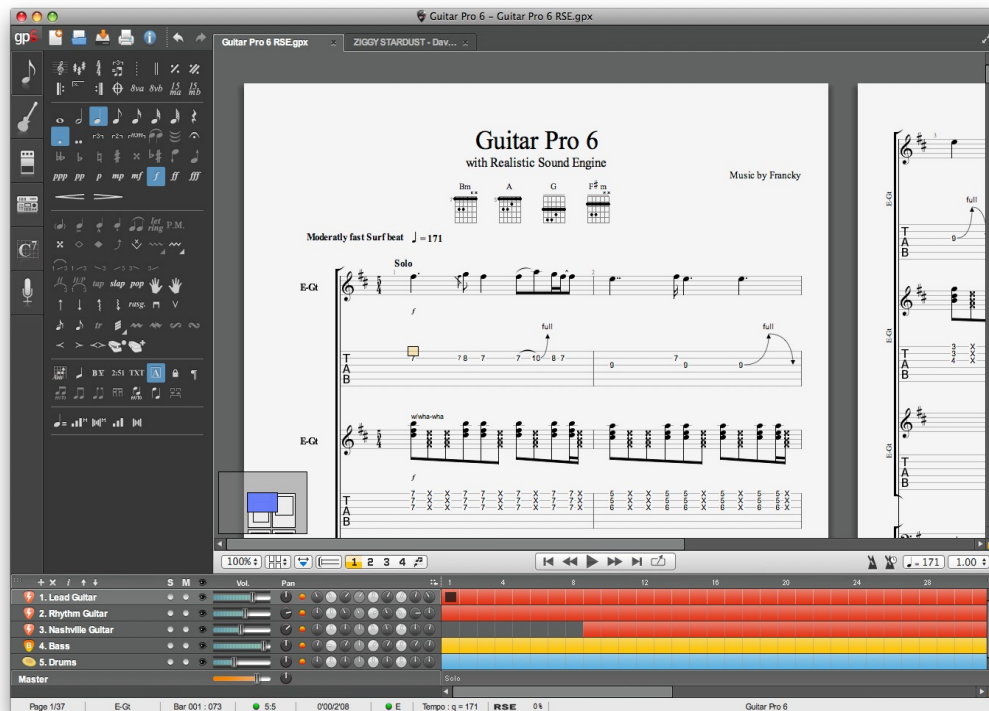


Figura 1: Screenshot do Guitar Pro (adaptado de [3])

Observamos na Figura 1 uma tablatura e partitura criada no Guitar Pro 6, onde à esquerda possuem os elementos musicais para construção de uma partitura. O Guitar Rig [5] apresentado na Figura 2, é outra aplicação vocacionada para guitarristas, simulando um pedal de efeitos no computador.



Figura 2: Screenshot da aplicação Guitar Rig (adaptado de [5])

O Guitar Rig realiza efeitos em tempo real com baixa latência no computador, como se de uma pedaleira real se tratasse. Uma aplicação com capacidade de criar tablatura para o

guitarrista a partir do que ele toca, iria permitir que o guitarrista não tivesse que gravar e ouvir constantemente o que tinha tocado para o poder reproduzir, como também iria ajudar no processo criativo da composição e improviso.

O objectivo do projecto consiste no desenvolvimento de uma aplicação que possibilite ao músico gravar o som da guitarra durante o processo de improviso, detectar as notas tocadas e automaticamente criar uma tablatura e/ou partitura. A captação/aquisição do sinal é feita em tempo real, mas não existem requisitos de tempo real no processamento e na apresentação das notas. Assim, conseguir-se-á uma aplicação capaz de auxiliar o guitarrista no seu processo de composição.

As aplicações existentes para guitarrista, em geral são divididas em dois tipos: aplicações para efeitos, como o Guitar Rig e para notação e composição musical. Comparativamente com aplicações de notação musical como Guitar Pro, a diferença seria que na aplicação o guitarrista só necessitava de exprimir com a guitarra toda a sua criatividade, e a aplicação automaticamente registava esse momento criativo.

Ao contrário do que acontece com o Guitar Pro, por exemplo, seria necessário que o guitarrista gravasse, ou tomasse nota passo a passo do que estava a tocar, o que iria de certa forma atrasar e perturbar todo o processo criativo. Com a solução proposta Guitar Solo Tracker, o guitarrista poderá ter a aplicação a ser executada e ir tocando a guitarra, e ao mesmo tempo que grava todo o som. O guitarrista poderia também estar a ensaiar com a banda e gravar todo o ensaio, para depois poder corrigir; isso ajudaria o guitarrista a guardar e registar qualquer momento de genialidade.

1.1 Soluções existentes

A nível de mercado, não existem produtos que possuam as mesmas funcionalidades, tanto ao nível de software como ao nível de hardware. O dispositivo de hardware mais próximo à solução que se pretende desenvolver, por requerer captura e identificação das notas tocadas na guitarra é o Roland VG-99 [6]. Este sintetiza o sinal vindo da guitarra eléctrica, captado por meio de captadores adaptados à guitarra e um conversor MIDI apresentado na Figura 3 [7].



Figura 3: Sintetizador Roland VG-99 (Adaptado de [6])

O Roland VG-99, apresentado na Figura 3 é responsável pela sintetização do sinal vindo da guitarra. Desta forma torna possível o guitarrista produzir o som com a guitarra e em seguida reproduzir esse som como se fosse produzido por um piano, por exemplo. Para captar o sinal da guitarra, este dispositivo possui um captador próprio que é adicionado à guitarra, e ligado a um conversor MIDI, como apresenta a Figura 4 [8].



Figura 4: Captadores para captar as notas da guitarra (adaptado de [8])

Os captadores apresentados na Figura 4, têm que ser presos na guitarra e estes convertem as vibrações das cordas em sinais MIDI. Este sistema é caro comparativamente com uma solução por software, dado que é necessário comprar os captadores, o conversor MIDI e o sintetizador.

1.2 Descrição geral da solução proposta

A solução desenvolvida neste trabalho tem o seguinte funcionamento:

A guitarra eléctrica é ligada ao computador através de uma placa de som. A aplicação instalada no computador vai captar o sinal enviado pela guitarra eléctrica e detectar a frequência do sinal enviado pela guitarra e com base na frequência detectada, é feito o mapeamento para notas musicais. Na aquisição do sinal da guitarra calcula-se a duração da nota para ser utilizada na criação de uma partitura. A Figura 5 apresenta o cenário de funcionamento e utilização da aplicação.

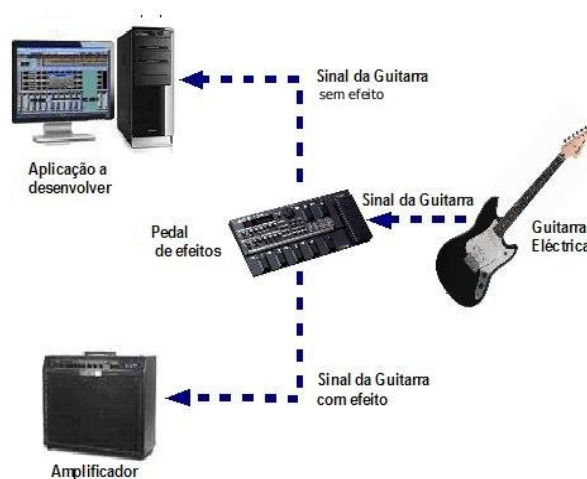


Figura 5: Esquema de funcionamento da aplicação

O sinal áudio da guitarra eléctrica poderá ou não passar por um pedal de efeito antes de ser ligado ao PC. Será realizado um estudo sobre a melhor opção entre ligar ao PC o sinal com ou sem efeito. Esta análise servirá para se definir um esquema final, depois de se obter como resultado a influência que os diversos efeitos de guitarra podem ter na detecção do sinal. No entanto, o sinal que será enviado para a aplicação será sem efeito. O facto do sinal ter algum efeito, pode influenciar na detecção correcta das frequências, porque muitos efeitos envolvem a alteração das harmónicas do sinal da guitarra.

A detecção das notas é feita mediante a utilização de algoritmos de detecção de frequência. Opta-se pela Fast Fourier Transform (FFT), após a análise de vários algoritmos. A apresentação do resultado ao guitarrista é feita num ficheiro de imagem representando uma tablatura que contém as notas identificadas pela aplicação.

A aplicação foi desenvolvida em ambiente Linux, com a distribuição Fedora 14, de 64 bit, usando as linguagens C/C++ e C#. Embora tenham sido desenvolvidas em Linux, as bibliotecas utilizadas suportam outras plataformas. A ferramenta de desenvolvimento utilizada foi o MonoDevelop [9] que também está disponível para Windows.

1.3 Organização do texto

O presente relatório está organizado em 7 Capítulos.

No Capítulo 2 realiza-se a introdução ao problema, nomeadamente com conceitos e terminologias associados à música (Capítulo 2, secção 2.1). Esta introdução à terminologia é importante para se compreender a ligação entre a música e o processamento de sinal, bem como para obter uma percepção mais detalhada do problema que se pretende resolver. É feita ainda no Capítulo 2, a introdução às notas musicais, a sua notação em pauta e a relação com frequências. A partir da secção 2.2 é descrito o equipamento que será utilizado, nomeadamente a guitarra eléctrica, a sua constituição, a forma de produção de som e notas musicais, e também a largura de banda associada. Na secção 2.3 descreve-se em pormenor o problema que se pretende resolver.

No Capítulo 3, é abordado o estudo e análise feita com algoritmos de detecção de frequência, onde são analisadas a FFT, Goertzel e auto-correlação [10]. São analisados os algoritmos estudados escolhendo-se o que melhor se adequa ao problema.

O Capítulo 4 descreve a solução proposta para o problema. Aborda-se a arquitectura da aplicação, e o esquema de funcionamento. A arquitectura da aplicação ilustra como a aplicação foi dividida em várias camadas funcionais, detalhando o tipo de funcionalidade de cada camada no contexto geral. O esquema de funcionamento relata como o utilizador final, o guitarrista, usa a aplicação, e como se interligam a guitarra eléctrica e o computador.

O Capítulo 5 apresenta os aspectos referentes ao desenvolvimento do projecto, as várias camadas e as classes que as constituem. Por cada classe é apresentada uma tabela que associa as classes aos ficheiros do projecto, e também o diagrama Unified Modeling Language (UML) de cada classe. Também é abordada neste Capítulo, na secção 5.2 a integração entre as duas componentes principais da aplicação desenvolvida, nomeadamente o *GSTTransporter* e a interface gráfica.

No Capítulo 6 reportam-se e analisam-se os resultados obtidos no funcionamento da aplicação. É analisada a precisão da detecção das notas e o desempenho da aplicação na execução da aplicação.

O relatório termina no Capítulo 7 com as conclusões e descrição de possíveis direcções de trabalho futuro que poderão melhorar o desempenho da aplicação bem como introduzir novas funcionalidades.

2 Formulação do problema

Neste capítulo é formulado o problema que se pretende resolver. São caracterizados todos os elementos que fazem parte do problema e desenvolvimento da solução. Introduzem-se conceitos relacionados com a composição musical e sua representação, bem como as características da guitarra eléctrica e outros equipamentos de hardware e software que foram utilizados no projecto.

2.1 Conceitos e terminologia associados à música

A música pode ser definida como um conjunto de sons organizados, ou a arte de exprimir sons. O som provém de vibração de um corpo e tem 4 propriedades; frequência, amplitude, timbre e duração. A frequência do som está relacionada com o maior ou menor número de vibrações por segundo do corpo que o produz. A intensidade do som é a potência com que se manifesta. O timbre é a qualidade do som, sendo que cada instrumento tem um timbre diferente. A duração é o tempo que o som dura até extinguir-se.

O som é representado como uma ou mais notas musicais; cada nota representa um tom ou semitom. No conjunto, existem 12 notas musicais, constituídas por 7 naturais e 5 acidentes. Tendo como exemplo o teclado do piano apresentado na Figura 6, as notas correspondem às teclas brancas e as pretas correspondem aos acidentes.

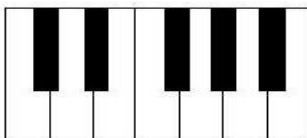


Figura 6: Teclado de piano, as teclas brancas são notas naturais e as pretas acidentes

Cada nota natural tem uma designação: Dó, Ré, Mi, Fá, Sol, Lá, Si. As notas musicais também podem ser representadas no formato angolosaxonico C, D, E, F, A, B, para as notas de Dó a Si, respectivamente, como apresenta a Figura 7.

Na Figura 7, os acidentes são simbolizados como sendo a nota natural aumentada ou diminuída de 1/2 tom com # (sustenido) ou *b* (bemol), respectivamente. Um sustenido eleva a nota natural a um semitom enquanto um bemol baixa a nota um semitom.

Embora existam 12 notas, um instrumento como o piano possui mais do que 12 teclas. O conjunto de 12 notas é chamado uma oitava, as 12 notas são repetidas em todas as oitavas. Num piano, o conjunto de 12 teclas, ou as oitavas, da esquerda para a direita correspondem a sons mais agudos. Assim as notas mais à direita têm sons mais agudos. A Figura 8 exemplifica uma oitava.

Em termos físicos, o som corresponde a uma vibração de determinada frequência. Consequentemente uma nota musical representa uma vibração (frequência), específica que a identifica como sendo um Dó ou uma das 12 notas. Com a variação das oitavas variam também os valores da frequência, sendo que as mais baixas formam sons graves e as mais altas sons agudos. A nível Europeu por convenção a nota (Lá) foi escolhida como frequência referência de 440 Hz, na terceira oitava. Se pretendemos obter a mesma nota

numa oitava acima (na quarta oitava), multiplicamos a frequência da nota proveniente da oitava anterior por 2.

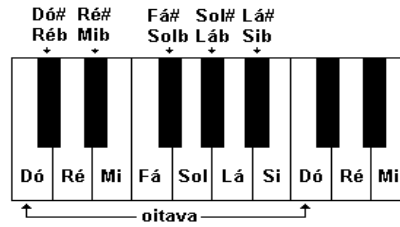


Figura 7: Exemplo de uma oitava no piano

Na Figura 7 podemos observar que uma oitava é contabilizada pelo conjunto das 12 notas. As frequências das 12 notas musicais formam uma progressão geométrica, cuja razão é de $2^{1/12} = 1,059$. Significa que partindo de uma nota base, podemos obter as frequências das notas seguintes, através de,

$$f = 2^{n/12} * 440 \text{ Hz} \quad (1)$$

em que n representa a distância que a próxima nota está da frequência referência. Também é possível calcular a nota correspondente a partir de uma frequência. Na equação (2), obtemos a distância que a frequência passada em f está da frequência referência. Dado que a frequência tomada como referência corresponde à nota Lá, notas à esquerda de Lá terão valor negativo, ou seja, uma distância negativa, enquanto as que estão do lado do direito terão valores positivos. Assim com a equação (2), tendo como centro a nota Lá, n pode ser mapeada para a nota correspondente na oitava certa.

$$n = 12 * \log_2(f/440) \quad (2)$$

Para que todas as notas estejam correctas é necessário que o instrumento esteja afinado, ou seja, que todas as notas correspondam de facto à frequência esperada. A Figura 8 [11] exemplifica o braço da guitarra com todas as notas, entre os 22 trastos.

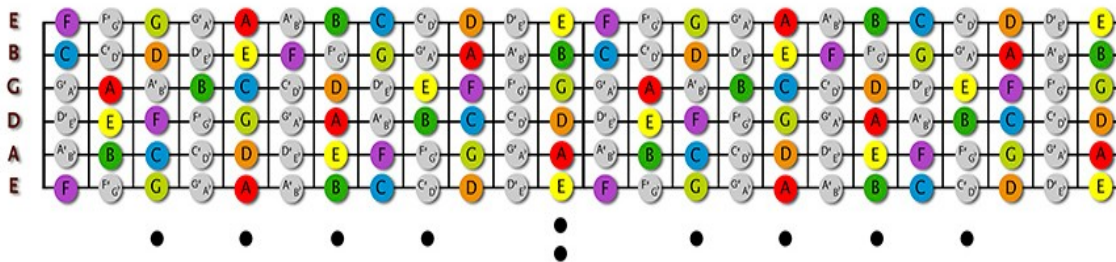


Figura 8: Exemplo do braço da guitarra com todas as notas (adaptado de [11])

No caso da guitarra, as notas estão separadas por trastos, e a distância de cada trasto é um semitom. Na Figura 9, cada linha representa uma corda na guitarra e cada coluna representa o trasto. Cada corda pressionada contra o trasto produz um semitom.

Define-se compasso como sendo a divisão do tempo em partes iguais. Cada compasso é dividido em partes de igual duração, chamadas Tempos [12]. O tempo define o andamento da música e é medido em Beat Per Minute (BPM). Define-se ritmo como a duração do som com o tempo dentro de um compasso, é a pulsação da música.



Figura 9: Metrônomo analógico (adaptado de [13])

Um metrônomo é um aparelho que serve para marcar o tempo da música, ou seja, determina a velocidade com que a música é tocada por todos os instrumentos. Quando uma banda toca a mesma música, ou mesmo quando alguém quer reproduzir uma música é necessário que se mantenha a velocidade certa, para dar o ritmo e compasso certo da música. A Figura 9 [13] apresenta um metrônomo analógico. O metrônomo analógico é mecânico, e composto por um pêndulo metálico com um peso deslizante, que é utilizado para regular o andamento. O metrônomo possui uma escala graduada em Beat Per Minute (BPM) por detrás do pêndulo. Quanto mais próximo estiver o peso do eixo, mais rápida a velocidade do pêndulo; quanto mais afastado, mais lenta será essa velocidade. O balançar do pêndulo para a esquerda e para direita, produz internamente um som *click*, e este faz mover a corda. O metrônomo digital é eletrônico e utiliza oscilador de quartzo para produzir a frequência de cada batida. A Figura 10 [14] apresenta um metrônomo digital.

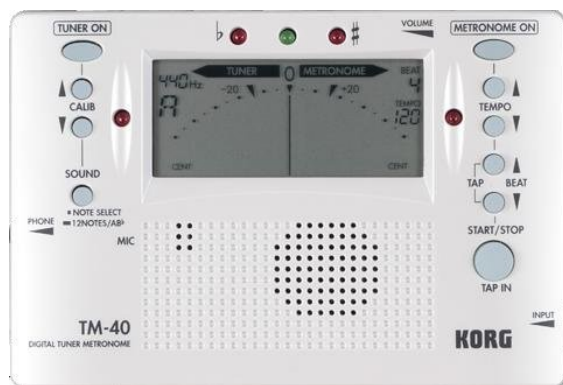


Figura 10: Metrônomo TM-40 (adaptado de [14])

Actualmente já existem metrônomos digitais como o TM-40 da KORG [14], cuja principal vantagem é poder alterar os tons de *click* para cada compasso, regular o volume do som, definir o número de BPM alinhados com o compasso.

Embora as notas estejam simbolizadas, não é possível representar todos os detalhes de uma música. Detalhes tais como o andamento da música, as notas que são tocadas em cada batida, os momentos de silêncio, ou seja, tudo que possibilite que alguém possa reproduzir a música tal qual quando foi criada. Para responder a essa necessidade foi desenvolvida notação musical, com a representação em pauta musical.

2.1.1 Notação musical

Para que se pudesse de alguma forma registar todos os aspectos de uma música, por forma que pudesse ser reproduzida na íntegra por quem pretendesse, foi desenvolvida uma notação musical, chamada pauta musical ou pentagrama.

A pauta musical, na Figura 11 é dividida por 5 linhas verticais e 4 espaços, que se chamam barra de divisão. À distância compreendida entre duas barras de divisão chama-se compasso. O sinal de repetição indica que o que estiver entre este sinal deve ser tocado mais uma vez. A fórmula de compasso determina o número de tempos em que se divide cada compasso; na Figura 11 é apresentada uma divisão de 4, compasso quaternário.



Figura 11: Elementos de uma pauta musical

Nas linhas horizontais e espaços são colocadas os símbolos que representam as notas. A armadura colocada em cima das linhas ou nos espaços, transforma a nota aí posta em sustenido ou bemol.

A pauta começa pela definição da clave, sendo esta usada para definir o posicionamento das notas. Existem 3 tipos de clave: Sol, Fá e Dó, como se apresenta na Figura 11.

As claves determinam onde começa a nota correspondente ao nome da clave. Observamos que os símbolos de clave de Fá e Dó têm alturas diferentes e possuem o símbolo (:) no final. A altura do símbolo determina que a linha entre os dois pontos representa a nota a qual a clave tem o nome. A clave de sol é mais utilizada para sons agudos, tais como, violino, guitarra flauta, entre outros. A clave de Fá é usada para sons graves, contra-baixo, baixo, tuba, entre outros.

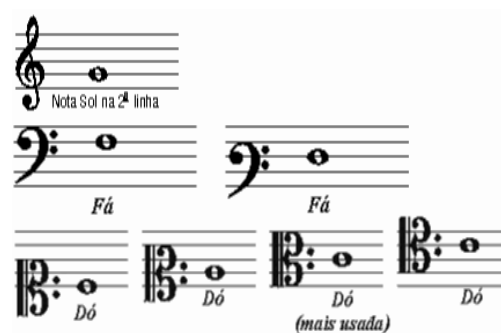


Figura 12: Clave de Sol, Fá e Dó

A clave de Dó é usada para sons médios como da viola, mas a sua aplicação é rara. Para se registrar os sons do piano é necessário o uso de duas claves, uma para notas altas e outra para notas baixas. Isso deve-se ao facto do piano possuir da esquerda para direita, tons graves a agudos, e estes tons variam a sua oitava dependendo do número de teclas, mas é considerado como Dó central o localizado na 3ª oitava. De baixo para cima representam-se os sons mais graves até aos mais agudos, tal como se apresenta na Figura 13.

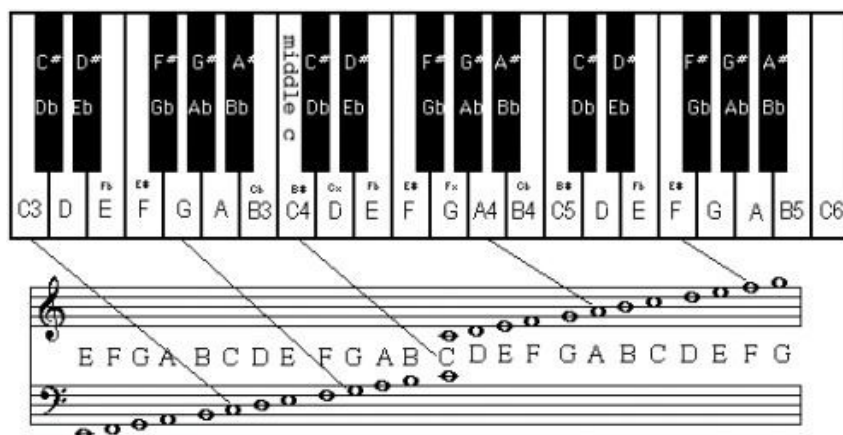


Figura 13: Notação musical para piano (adaptado de [15])

A Figura 14 exemplifica uma escala possível num piano em partitura, mapeada para as teclas correspondentes no piano. Neste caso podemos observar na parte inferior o símbolo de clave de Fá e na parte superior a clave de Sol.

2.1.2 Representação das notas

Para representarmos as notas na pauta, estas são colocadas nas linhas, e o seu valor depende da clave que está a ser usada. Também, para além das notas, são representados os tempos dos silêncios ou pausas. A representação das notas e pausas é feita de diferentes formas que representam o seu valor a nível temporal. A Figura 14 [16] apresenta uma Tabela que contém os símbolos que representam as notas e pausas numa pauta musical. O número relativo a cada símbolo, indica o tempo de duração da nota dentro do primeiro compasso ou o número de vezes que esta nota cabe num compasso.















Número Relativo	Nota	Pausa	Nome
1			Semibreve
2			Mínima
4			Seminima
8			Colcheia
16			Semicolcheia
32			Fusa
64			Semifusa

Figura 14: Tabela com figuras musicais para representação de notas e pausas (Adaptado de [16])

Observa-se na Figura 15 que para cada nota está associada uma pausa com a mesma duração, a alternâncias entre notas musicais e pausas, produzindo a música. A Figura 16 [17] exemplifica a aplicação destes símbolos em diferentes compassos.

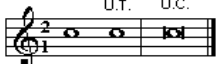






Compasso Binários	Compasso Ternários	Compasso Quaternários
U.T. U.C. 	U.T. U.C. 	U.T. U.C. 
		
		
		
		
		

Figura 15: Exemplos de aplicação das figuras em diferentes compassos (adaptado de [17])

Na Figura 15 observam-se os símbolos em diferentes compassos, temos 3 tipos de compassos diferentes; binário, ternário e quaternário. Verificamos que com base no compasso e divisão só podemos ter duas, três ou quatro notas, para os compassos binário,

ternário e quaternário respectivamente. Cada uma destas figuras musicais está relacionada entre si; a Figura 16 [18] apresenta a relação entre as figuras musicais.

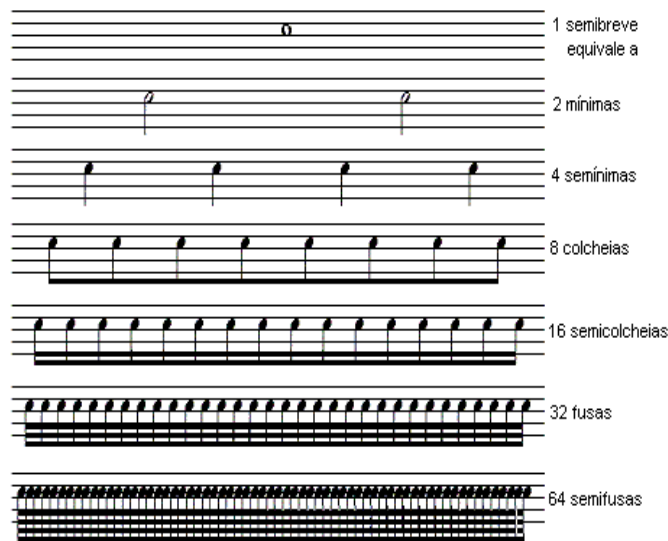


Figura 16: Relação entre as figuras musicais (adaptado de [18])

Observa-se na Figura 16 que uma semibreve equivale a duas mínimas, uma mínima equivale a 2 semínimas. Em geral podemos dizer que começando das figuras musicais de maior duração a semibreve, existe uma relação exponencial de base 2 entre as várias figuras musicais.

No caso da guitarra, o instrumento de foco neste projecto, existe outra forma de registar as notas para quem não tenha conhecimento de leitura de pautas musicais. Para guitarristas, anotam-se as notas da guitarra na forma de tablatura, tal como se apresenta na Figura 17.

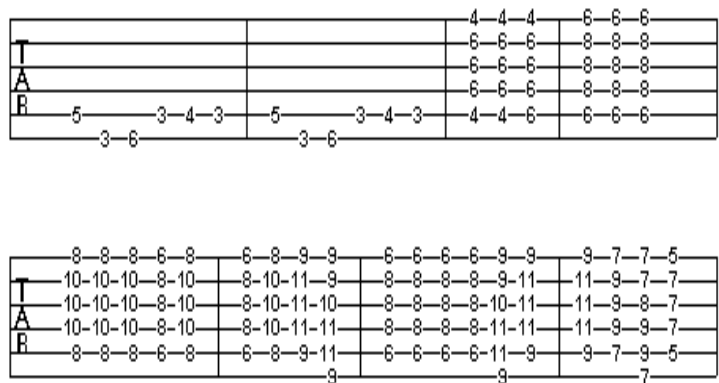


Figura 17: Exemplo de tablatura para guitarra

Na tablatura apresentada, as linhas horizontais representam as cordas que os instrumentos têm e os números representam os trastos que devem ser pressionados para reproduzir a música.

A tablatura é uma forma de notação musical, que indica onde colocar os dedos no instrumento em vez de informar a nota. Esta forma de notação pode ser utilizada em todos os instrumentos que usem trastos como a guitarra, guitarra-baixo e banjo, entre outros. Em adição, na tablatura é possível para o caso da guitarra, adicionar notação para alguns efeitos

como *slide* [19], *bending* [20], *tremolo* [21] entre outros, tal como exemplificado na Figura 18 [22].

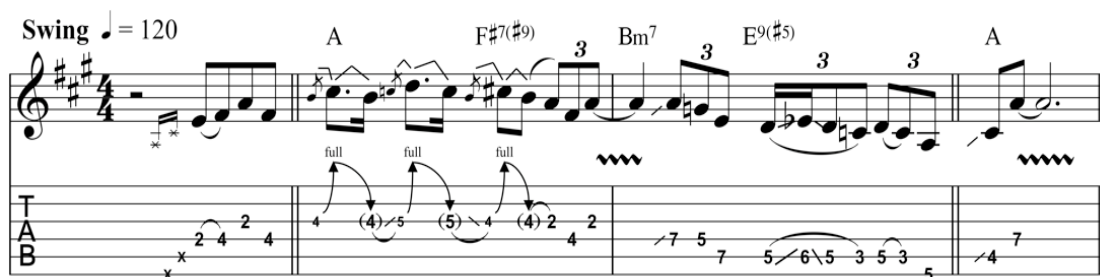


Figura 18: Exemplo de tablatura com efeitos ou técnicas (adaptado de [22])

Podemos observar na Figura 18 uma partitura com a tablatura associada, esta forma de representação é comum nas músicas para guitarra, pois permite guitarristas que não saibam ler pautas musicais reproduzirem a música. Na Figura 18 é possível observar os números que simbolizam os trastos que são pressionados na guitarra e alguns destes possuem outros símbolos associados. Estes símbolos, representam técnicas e efeitos tocados na guitarra. As Figuras 19 e 20 exemplificam alguns símbolos utilizados para representação de técnicas ou efeitos com a guitarra.

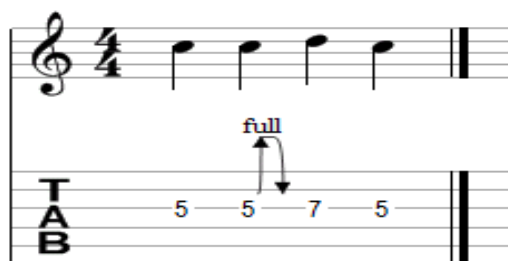


Figura 19: Exemplo de representação de Bend Up (adaptado de [23])

Na Figura 19 apresenta o símbolo para técnica de *bend*, a seta a apontar para cima significa *bend up* e a seta apontar para baixo *bend down*. A técnica de *bend* [20], consiste em empurrar a corda para cima ou para baixo, fazendo com que a nota varie de meio a um tom. A inscrição de *full* em cima do símbolo, significa que a corda é empurrada para cima até a nota variar um tom. A seta que aponta para baixo, quando associada a seta de *bend up*, significa que depois de efectuar o *full bend* o guitarrista volta a por a corda na posição inicial em vez de a soltar.



Figura 20: Exemplo da técnica Legato Slide (adaptado de [23])

A técnica de *Legato Slide* [19] consiste em tocar uma nota num trasto e deslizar o dedo sobre a corda, até ao próximo trasto, fazendo a nota variar a distância entre o primeiro trasto tocado e o último. Existem diversos símbolos utilizados na tablatura para representar diferentes técnicas; alguns podem ser consultados em [23].

2.2 Guitarra eléctrica e equipamentos

A aplicação será desenvolvida para ser utilizada com uma guitarra eléctrica. O uso da guitarra eléctrica impõe uma limitação que existe somente pelo facto de se poder garantir melhor qualidade na captura do som, dado que esta é feita directamente da guitarra para um amplificador ou placa de som. No entanto pode ser aplicada também a guitarras electro-acústicas ou mesmo guitarras clássicas, por meio de microfone. A guitarra eléctrica, como é constituída pelos elementos indicados na Figura 21 [24].



Figura 21: Guitarra eléctrica
(adaptado de [24])

- | | |
|-------------------------------|------------------------------------|
| 1. Mão ou paleta ou headstock | 11. Potenciómetros |
| 2. Pestana | 12. Cavalete (ou ponte) |
| 3. Tarraxas ou cravelha | 13. Protector de tampo (ou escudo) |
| 4. Trastes/Trastos | |
| 5. Tirante ou Tensor | |
| 6. Marcação | |
| 7. Braço | |
| 8. Tróculo (Junta do braço) | |
| 9. Corpo | |
| 10. Captadores | |

A guitarra eléctrica apresentada na Figura 21, possui um corpo em madeira maciça, ao contrário de uma guitarra clássica que possui um corpo oco de propagação acústica. Por este motivo o som da guitarra eléctrica natural é baixo, sendo necessário a utilização de um amplificador. O tipo de madeira utilizada no corpo, captadores, o material dos trastos, são determinantes no timbre e tom da guitarra eléctrica. No corpo da guitarra eléctrica estão incrustados elementos electrónicos como o potenciómetro e captadores que permitem a saída para ligação a um amplificador áudio.

2.2.1 Produção de som da guitarra eléctrica

A guitarra produz som fazendo vibrar as cordas que estão presas no braço ao corpo da guitarra. Quando a guitarra é tocada, produz um sinal de saída composto por frequências tais que correspondem a notas musicais. Pressionando a corda entre os trastos existentes no braço, diminuímos o seu comprimento fazendo com que a frequência da corda seja maior, quanto mais se próximo da ponte se estiver, produzindo diferentes sons. Quando os trastos são premidos mais próximos do corpo da guitarra, como a distância é menor entre a secção premeida e a solta, a vibração da corda produz um som mais agudo.

A guitarra eléctrica possui exactamente o mesmo princípio de funcionamento de uma guitarra normal (acústica); o que diferencia é o modo como o som é captado e propagado.

Na guitarra eléctrica, existem dispositivos designados por captadores. Estes dispositivos dispostos no corpo da guitarra, por baixo das cordas, captam a vibração da corda que faz variar o campo electromagnético da bobina que envolve o íman, gerando uma corrente que é enviado para um amplificador. Para que o som da guitarra eléctrica seja audível é necessário ligar a um amplificador de som, dado que não possui um corpo oco como a acústica e a clássica. A Figura 22 [25] apresenta um captador para guitarras eléctricas.



Figura 22: Captador de guitarra(adaptado de [25])

Podemos observar na Figura 20 um exemplo de um captador passivo. A diferença entre o captador passivo e activo está relacionada com o facto do captador activo ser alimentado por uma pilha introduzida no corpo da guitarra. A Figura 23 [25] exemplifica o funcionamento básico dos captadores na guitarra eléctrica.

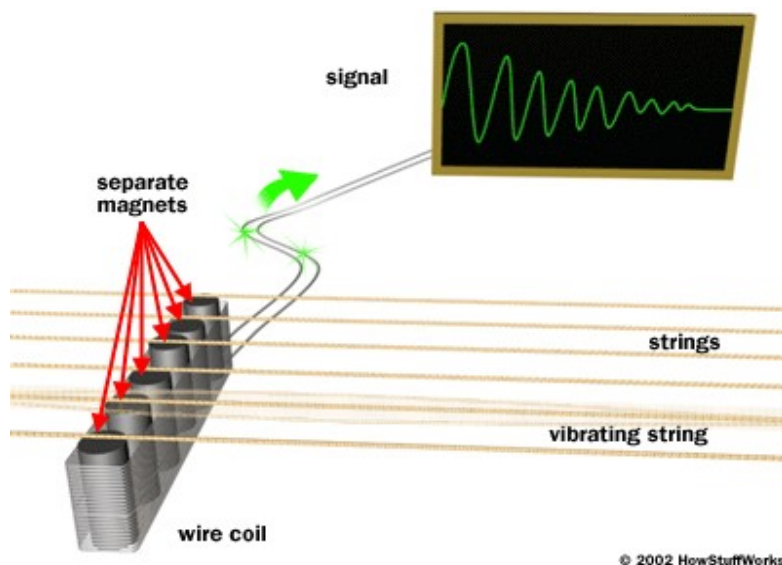


Figura 23: Funcionamento dos captadores (adaptado de [25])

Observamos na Figura 23 que os captadores são colocados por baixo das cordas para captar a vibração destas. A vibração é captada pelos ímanes permanentes e convertida em sinal eléctrico. A Figura 24 [26] exemplifica a forma como funciona internamente um captador passivo.

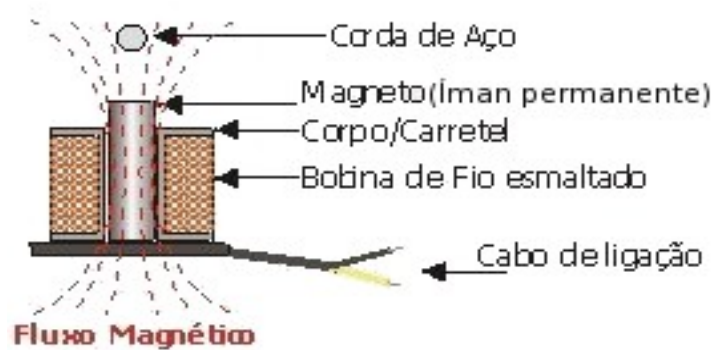


Figura 24: Esquema de um captador magnético (adaptado de [26])

O princípio geral de produção do som é o seguinte: a variação do campo magnético é convertida em tensão eléctrica que quando amplificada produz som. A guitarra pode ter cordas de diferentes materiais, sendo normalmente de aço, níquel e cobre. Cada corda tem determinada espessura para produzir a vibração correcta. As diferentes espessuras de cada corda, influenciam no tom e a oitava que pode produzir, mas não influenciam as harmónicas das notas.

Na saída da guitarra, o sinal analógico tem amplitude de 100 mV ou 1 V . A variação da amplitude do sinal depende do tipo de captador. A principal diferença entre estes está na definição do som obtido pelos dois tipos de captadores e na amplitude na saída do sinal, sendo que os captadores activos conduzem a maior gama dinâmica de amplitude no sinal.

O sinal à saída da guitarra eléctrica é ligado a um amplificador de forma a tornar-se audível. Muitas vezes, entre a guitarra eléctrica e o amplificador é ligado um processador de efeitos, tais como *delay*, *distorção*, *chorus* entre outros.

2.2.2 Escalas e frequências da guitarra eléctrica

A guitarra eléctrica tem entre 22 a 24 trastos, e tipicamente 6 cordas. Começando do topo para baixo, cada corda tem uma espessura/secção diferente; as de maior secção produzem som mais grave e as de menor secção produzem som mais agudo.

Cada corda, tocada em aberto (tocada sem se premir qualquer trasto), corresponde a uma nota musical, *Mi (E)*, *Lá (A)*, *Ré (D)*, *Sol (G)*, *Si (B)* e *Mi (E)*, de cima para baixo no braço da guitarra, respectivamente. Premindo os trastos ao longo do braço em cada corda, permite ir gerando as notas seguintes que cada corda possui.

A guitarra eléctrica, produz frequências entre 82,41 Hz e 1174 Hz. Assim, possui no máximo 3 oitavas completas, como mostra a Tabela 1. Cada corda produz um conjunto de frequências quando tocadas e pressionadas nos trastos ao longo do braço da guitarra. No processo de criação da tablatura, será necessário ter em conta a fisionomia das mãos. Este requisito é importante para otimizar a apresentação das notas no braço da guitarra, permitindo que quem toque o instrumento não tenha notas fora do alcance das mãos e faça menos movimentos desnecessários. Por exemplo, ao tocar mais do que uma nota em simultâneo, com o dedo indicador no segundo trasto não é possível com os outros dedos

pressionar o décimo trasto. Define-se como acorde a escrita ou execução de duas ou mais notas musicais em simultâneo.

OITAVAS								
	NOTAS	1	2	3	4	5	6	7
F	A		27,5	55	110	220	440	880
R	Bb		29,14	58,27	116,54	233,08	466,16	932,33
E	B		30,87	61,74	123,47	246,94	493,88	987,77
Q	C		32,7	65,41	130,81	261,63	523,25	1046,5
U	C#		34,65	69,3	138,59	277,18	554,37	1108,73
Ê	D		36,71	73,42	146,83	293,66	587,33	1174,66
C	D#		38,89	77,78	155,56	311,13	622,25	1244,51
I	E	20,6	41,2	82,41	164,81	329,63	659,26	1318,51
A	F	21,83	43,65	87,31	174,61	349,23	698,46	1396,91
S	F#	23,12	46,25	92,5	185	369,99	739,99	1479,98
	G	24,5	49	98	196	392	783,99	1567,98
(Hz)	G#	25,96	51,91	103,83	207,65	415,3	830,61	1661,22

Tabela 1: As colunas representam as oitavas e as linhas a notas musicais

As frequências na banda de 82.41 Hz a 1174.66 Hz correspondem a uma guitarra eléctrica de 24 trastos, enquanto de 30 Hz a 359.23 Hz são frequências de uma guitarra baixo.

Para os testes experimentais será usada uma guitarra eléctrica Yamaha EG112 com 22 trastos. No entanto, o tipo e modelo de guitarra eléctrica não tem qualquer influência na aplicação.

2.2.3 Hardware e software

A aplicação desenvolvida tem suporte multi-plataforma, nomeadamente para os sistemas operativos Linux e Windows XP/Vista/7 com processamento paralelo em computadores com processadores multi-core. O suporte a vários sistemas operativos, visa a dar oportunidade ao guitarrista de escolher o sistemas operativos de sua preferência.

Uma opção para suportar a portabilidade seria a utilização da linguagem JAVA, mas não seria tão imediato tirar partido de bibliotecas já existentes em C/C++, nomeadamente da FFT para processamento de sinal e seria mais lento. A vantagem de utilização da linguagem C/C++ em relação ao JAVA, prende-se principalmente na eficiência da realização de componentes críticos como processamento áudio, a possibilidade de inclusão de algumas optimizações usando paralelismo e instruções Single Instruction Multiple Data (SIMD) como Streaming SIMD Extensions (SSE) e SSE2. A linguagem C/C++ proporciona maior controlo e proximidade com o hardware.

Características	Computador 1	Computador 2
Sistema Operativo	Fedora 13 x86_64	Windows 7 32 bits
Processador	Intel Core i7	Intel Dual Core
Memória	4 Gb	2 Gb
Disco duro	1.5 Tb	100 Gb

Tabela 2: Características dos computadores de desenvolvimento

A aplicação foi desenvolvida e testada em computadores com as características apresentadas na Tabela 2.

2.3 Problema a resolver

O problema a resolver consiste em conseguir de forma eficiente detectar as frequências de notas recolhidas de uma guitarra eléctrica e estimar a duração destas, para construir uma tablatura, sem que se percam as notas seguintes tocadas na guitarra. O processamento não tem que ser realizado em tempo real mas a aquisição é em tempo real. A possível perda de notas está relacionada com o número de notas que o guitarrista pode tocar numa batida de metrónomo, considerando n BPM, onde $n \in \{46, \dots, 180\}$.

A detecção de frequências é um problema comum na área de processamento digital de sinal, para o qual existem algoritmos como Goertzel [27], FFT [27] e auto-correlação [10]. Cada um destes algoritmos possui características que serão determinantes para utilização numa solução. Estes algoritmos foram analisados num trabalho prévio efectuado no âmbito da unidade curricular de Processamento de Sinal em Tempo Real (PSTR) e são apresentados no capítulo 3.

Outra componente do problema é a detecção de técnicas executadas enquanto o guitarrista toca. No caso de um guitarrista solo, existem técnicas como *leggato*, *shredding* [2], assim como *speed picking* solo, em que são tocadas várias notas por batida de forma sequencial; neste caso é difícil detectar todas as notas apenas com a audição do sinal à medida que este vai sendo produzido.

3 Técnicas de processamento de sinal para detecção de frequência

Neste capítulo analisam-se várias técnicas de processamento de sinal para estimação de frequência e duração da nota. O objectivo é encontrar o algoritmo de detecção de frequências que mais se adequa à detecção das notas da guitarra. Como métricas de selecção pretende-se escolher o algoritmo que possua a melhor precisão, menor complexidade na implementação da solução e no número de multiplicações.

3.1 Algoritmo FFT

A Fast Fourier Transform (FFT) [10] é um algoritmo eficiente para calcular a Transformada de Fourier Discreta ou Discrete Fourier Transform (DFT) e sua inversa. A DFT um sinal discreto de N amostras $\{x(0), x(1), x(2), \dots, x(N-1)\}$ espectro, $X[k]$, em N pontos de amostragem para $k = 0, 1, 2, 3, \dots, N-1$, em que k é o índice em frequência, n o índice no

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\left(\frac{2\pi k}{N}\right)n}, k=0, 1, \dots, N-1 \quad (3)$$

tempo e $X[k]$ são os coeficientes da DFT. A DFT gera N coeficientes de $X[k]$ para $k = 0, 1, 2, 3, \dots, N-1$. A resolução em frequência da DFT a N pontos é dada por:

$$\Delta = \frac{f_s}{N}, \text{ onde } f_s \text{ é a frequência de amostragem} \quad (4)$$

Podemos então calcular uma frequência f_k (em Hz) correspondendo ao índice k .

$$f_k = k \Delta = \frac{k f_s}{N}, k=0, 1, \dots, N-1 \quad (5)$$

De acordo com o ritmo de Nyquist, $f_s/2$ corresponde ao índice de frequência $k = N/2$, em que $|X[k]|$ é uma função par de k , porque $x[n]$ é um sinal real. Assim sendo, só é necessário amostrar o espectro onde k varia entre 0 e $N/2$. Através da aplicação da FFT e um número adequado de amostras N é possível detectar todas as frequências da guitarra.

A FFT apresenta complexidade logarítmica de cálculo, tornando-se muito interessante para várias aplicações. Para um sinal com N pontos, realizam-se

$$N_{op} = N \log_2(N) \quad (6)$$

multiplicações complexas. A DFT, sem optimização, realiza N^2 multiplicações complexas. Cada multiplicação complexa corresponde a quatro multiplicações reais e duas adições reais.

Para o problema a ser resolvido não existe a necessidade de determinar cada nota com exactidão, sendo possível encontrar um número de pontos adequado à detecção de todas as frequências com o menor erro médio possível.

3.2 Algoritmo Goertzel

O algoritmo de Goertzel [27] detecta a presença de uma componente de frequência. A transformada Z associada ao sistema que realiza o algoritmo de Goertzel é dada por

$$H_k(z) = \frac{1 - W_N^k z^{-1}}{(1 - W_N^{-k} z^{-1})(1 - W_N^k z^{-1})} = \frac{1 - e^{j2\pi k/N} z^{-1}}{1 - 2 \cos(2\pi k/N) z^{-1} + z^{-2}}. \quad (7)$$

A FFT realiza a DFT de forma eficiente amostrando várias frequências do espectro. O algoritmo de Goertzel detecta apenas frequências específicas e implementa uma equação às diferenças de segunda ordem, correspondente ao diagrama de blocos da Figura 25.

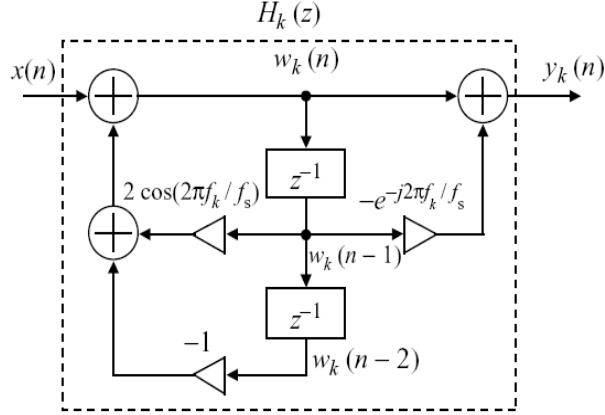


Figura 25: Diagrama de bloco do filtro ordem II de Goertzel

Para a detecção de frequência é necessário calcular a magnitude do coeficiente $X(k)$, dada por

$$|X(k)|^2 = \omega_k^2(N-1) - 2 \cos(2\pi f_k / f_s) \omega_k^2(N-2) + \omega_k^2(N-2) \quad (8)$$

em que N é o número de amostras de $x[n]$.

Para uma dada frequência a ser detectada, temos que calcular o valor do índice k que corresponde à frequência, e verificar neste índice a magnitude do sinal para se poder afirmar que o sinal contém a frequência esperada. O algoritmo de Goertzel ao contrário da FFT não necessita que o número de pontos seja potência de 2 o que dá maior flexibilidade para aumentar ou diminuir o número de pontos. A equação (9) serve para determinar o índice k para uma determinada frequência de interesse, onde $freq$ é a frequência de interesse, f_s é frequência de amostragem e N número de pontos.

$$k = \frac{freq}{f_s} * N \quad (9)$$

A maior problemática relativamente ao objectivo, reside no facto de as frequências que pretendemos detectar terem valor baixo. Para podermos estimar todas as frequências, considerando resolução de frequência de 2 Hz e as frequências de amostragem descritas, seria necessário um N igual aos apresentados na Tabela 3, usando as frequências de amostragem correspondentes. Ao escolher valores elevados de N , tem-se o problema de

falta de resolução no tempo, ou seja, é necessário adquirir sinal durante um tempo excessivo que pode ser muito superior ao tempo de duração da nota.

Freq. Amostragem	$N = F_s / 2 \text{ Hz}$	Resolução no Tempo (s)
44100	22050	0,5
22050	11025	0,5
11025	5512,5	0,5

Tabela 3: Número de pontos com base em F_s com resolução de 2 Hz

Podemos observar na Tabela 3 que os valores de N são muito grandes o que pode afectar o tempo-real na pesquisa pela frequência a detectar e também aumentar o recursos computacionais. No PC podemos tentar várias estratégias para otimizar a pesquisa pela frequência a determinar e diminuir N , para utilização deste algoritmo.

Uma estratégia seria ter um número de pontos fixos para cada nota que se pretende detectar; neste caso teríamos 47 *buffers* no total. O número total de *buffers* a ser utilizado não proporciona uma solução óptima de implementar e nem a nível de consumo de recursos computacionais, dado que seria sempre necessário fazer a cópia do sinal para os 47 *buffers*.

A segunda estratégia seria ter um *buffer* suficientemente grande para se obter todas as notas, para otimizar a pesquisa, seriam utilizados previamente filtros para filtragem por oitava. Assim sendo, antes do sinal passar pelo algoritmo, passavam por filtros e estes com base na oitava iriam definir os pontos de pesquisa no *buffer*. Neste caso, estaríamos sempre a pesquisar no máximo por 12 notas e no mínimo 5, atendendo ao número de escalas que a guitarra possui.

A problemática desta solução está no tamanho do *buffer*, que poderá influenciar no tempo de cópia e leitura das próximas amostras tocadas pelo guitarrista. Para resolver este problema, podemos utilizar as técnicas de *double buffering*, e *multi-threading*. Esta estratégia só teria melhor aplicação para uma frequência de amostragem mais baixa, baseando na Tabela 3.

A terceira estratégia consiste em ter um único *buffer* de tamanho N , mas para cada nota é dada uma dimensão diferente e menor que N . Neste caso a pesquisa por cada nota seria limitada ao tamanho associado a cada nota. No entanto existe a possibilidade de termos índices de notas coincidentes, o que poderá dificultar a identificação da nota. As notas coincidentes ocorrem quando as mesmas notas mas em oitavas diferentes são calculadas na mesma posição no *buffer*, criando uma indefinição de que nota está presente. Para otimizar a pesquisa, também usamos filtros para determinar a oitava a que a nota pertence, por forma a pesquisar somente pelas notas ou frequências pertencentes à oitava.

Em resumo, para aplicar o algoritmo de Goertzel para estimação das frequências seria necessário aplicar estratégias diferentes para resolver os problemas encontrados no estudo. Estas estratégias não representam uma melhor solução para o problema que se pretende resolver, pois envolvem maior complexidade no desenvolvimento, e não se consegue o objectivo que é a redução do número de pontos de forma efectiva.

3.3 Auto-correlação

O algoritmo de auto-correlação em processamento de sinal determina a relação existente entre o sinal e ele próprio ao longo do tempo. A auto-correlação representa o grau de proximidade entre a versão original do sinal de entrada e uma versão desfasada do mesmo sinal [28] [29].

Dado um sinal $x(t)$, a auto-correlação do sinal contínuo por si próprio com um desfasamento de τ é dado por

$$R_{xx}(\tau) = (x(t) * \bar{x}(-t))(\tau) = \int_{-\infty}^{\infty} x(t+\tau) \bar{x}(t) dt = \int_{-\infty}^{\infty} x(t) \bar{x}(t-\tau) dt, \quad (10)$$

onde, \bar{x} corresponde a complexo conjugado do sinal $x(t)$ e $(*)$ representa a convolução. Com um sinal discreto a expressão da auto-correlação de x com desfasamento de k é dada por

$$R_{xx}[k] = \sum_n x[n] \bar{x}[n-k]. \quad (11)$$

A função de auto-correlação apresenta as seguintes propriedades:

- Simetria:

$$R_f(-\tau) = R_f(\tau)$$

- A auto-correlação de uma função periódica de período t , é também periódica com mesmo período.
- O maior valor de auto-correlação encontra-se em $\tau = 0$.

A auto-correlação pode ser realizada de forma eficiente através do cálculo da FFT e IFFT pelo teorema de Wiener–Khinchin, como apresenta nas seguinte expressões:

$$F_R(f) = FFT(X(t)) \quad (12)$$

Dado $F_R(f)$ é igual a FFT do sinal de entrada $X(t)$, como apresentado em (12), multiplicado por $\hat{F}_R(f)$ seu conjugado (13), resultando na função densidade espectral de potência

$$S(f) = F_R(f(t)) \hat{F}_R(f(t)) \quad (13)$$

podemos determinar então a auto-correlação calculando a Transformada de Fourier Inversa de $S(f)$ apresentada em (14)

$$R(\tau) = IFFT(S(f)). \quad (14)$$

Para estimar a frequência usando este algoritmo é necessário calcular a correlação entre o sinal de entrada no tempo 0 com o mesmo sinal desfasado no tempo T . Como o sinal é periódico, o valor de T a ser utilizado seria igual ao valor de resolução em frequência que se

pretende ter, e o número de pontos necessário para estimar toda banda de frequência da guitarra eléctrica. Depois de calculada a auto-correlação é feita uma pesquisa pelo maior valor energia no espectro, e retirado o índice. Este índice indica que período tem a frequência fundamental do sinal de entrada. Podemos calcular assim a frequência tendo o índice do pico máximo utilizando (15) e (16)

$$\Delta t = N * T_s = \frac{N}{F_s}, \text{ onde } T_s \text{ período de amostragem.} \quad (15)$$

Em (15) calculamos a resolução no tempo, onde N é o número de pontos, T_s o período e F_s a frequência de amostragem. Com o valor do índice calculamos o período dado por

$$t_f = \frac{\Delta t}{k}, \quad (16)$$

onde t_f é o período da sinal de entrada, k o índice do primeiro pico máximo e Δt a resolução em frequência. Para obter o valor em frequência aplicamos (17) e calculando a inversa do do período t_f .

$$f_f = \frac{1}{t_f} \quad (17)$$

Ao contrário da FFT, a auto-correlação permite estimar uma frequência no domínio do tempo, identificando a frequência fundamental e suas harmónicas. A estimação da frequência usando a auto-correlação também pode ser aplicada para o problema a resolver, e existe uma forma eficiente de calcular a auto-correlação usando a FFT e IFFT. No entanto, dado que podemos simplesmente utilizar a FFT para fazer a estimação da frequência, não há necessidade de fazer a operação inversa.

3.4 Análise comparativa das soluções

No estudo inicial, na unidade curricular de PSTR, foram analisadas quais seriam os melhores algoritmos para detecção de frequências, e também os parâmetros óptimos para o número de amostras a considerar, a resolução em frequência, a frequência de amostragem. Foi efectuada uma prova do conceito em Matlab para testar os vários algoritmos.

No caso da FFT o problema está sempre na resolução de frequência necessária para detectar as notas, e este problema também é partilhado com auto-correlação e Goertzel. Tendo em conta que a aplicação não necessita de processamento de sinal em tempo real a questão essencial é não perder as amostras vindas da guitarra na fase de captura do sinal.

O uso de algoritmo de Goertzel [10] poderia ser interessante para determinar uma frequência específica, mas é muito comum na guitarra tocar acordes, e por este motivo é necessário um estudo relativamente a estratégia em usar Goertzel. Também com este algoritmo podíamos tentar usar vários vectores de amostras para detectar todas as notas existentes no instrumento. Esta solução no entanto poderia ser muito custosa, dado o número de notas existentes na guitarra eléctrica e também a dificuldade em detectar acordes.

A auto-correlação determina a frequência fundamental de uma nota. Após determinar a frequência fundamental, uma estratégia a usar seria fazer uso de filtragem para detectar a escala a que a nota pertence.

A maior problemática em usar estes algoritmos é o número de pontos necessários para detectar as frequências mais baixas, porque a resolução de frequência é muito baixa para a frequência de amostragem que o sinal estaria a ser amostrado numa aplicação no PC.

A análise da frequência de amostragem, foi efectuada porque seria determinante para obter o número de pontos necessário ao processamento áudio. Em trabalhos de áudio profissional a frequência de amostragem utilizada é de 44100 Hz ou superior. No entanto é necessário analisar a razão entre a máxima e a mínima frequência que poderiam obter em relação à frequência máxima e mínima que a guitarra eléctrica pode produzir, para determinar qual será a frequência de amostragem escolhida. A Tabela 4 apresenta a razão entre a frequência de amostragem e as frequência mínima e máxima produzidas pela guitarra eléctrica.

Freq. Amostragem (Hz)	Fs/ 82.41	Fs / 1174.66
44100	537,8	37,56
22050	268,9	18,78
11025	134,45	9,39
8000	97,56	6,81

Tabela 4: Razão entre a frequência mínima 82,41 Hz e máxima 1174,66 Hz

A razão de frequência com o valor mais baixo está para a frequência de amostragem de 8000 Hz ou de 11025 Hz. Dado que a máxima frequência da guitarra eléctrica é de 1174 Hz, uma amostragem de 8000 Hz é suficiente para se poder detectar todas as notas da guitarra.

Para cada um dos algoritmos de estimação de frequência era necessário ter diferentes abordagens em relação ao processamento áudio, tendo em conta conseguir detectar notas, estimar a duração das notas, e também a capacidade de detectar acordes.

Uma análise comparativa da complexidade no número de multiplicações entre os algoritmos é apresentada na Tabela 5, onde podemos verificar qual tem menor complexidade.

Algoritmos	Número de Multiplicações
DFT	N^2 complexas
FFT	$N * \log_2(N)$ complexas
Goertzel	N reais+ 1 complexa
Auto-correlação	$N * \log_2(N) * 2 + N$ complexas

Tabela 5: Complexidade dos algoritmos

Nos 3 algoritmos o problema reside sempre na resolução de frequência necessária para detectar as notas, e o facto da frequência de amostragem ser elevada para aplicar qualquer um dos algoritmos. Para reduzir o número de pontos, é necessário reduzir a frequência de amostragem. Uma técnica utilizada para reduzir a frequência de amostragem é a decimação [27] no tempo.

3.5 Filtragem e decimação

A decimação [27] corresponde à diminuição da frequência de amostragem de um sinal amostrado a uma taxa superior a de Nyquist. No entanto se um sinal for amostrado na taxa de Nyquist mas a sua largura de banda for reduzida por um filtro, a sua frequência de amostragem pode ser reduzida por decimação, evitando o fenómeno de *aliasing* [27].

A decimação no tempo é aplicada para reduzir a frequência de amostragem. Consiste em recolher $d*k$ amostras do sinal amostrado, onde $d \in \{1, \dots, \infty\}$ corresponde ao factor de decimação e $k \in \{0, \dots, N-1\}$ corresponde ao índice da amostra no *buffer* e N número de pontos do *buffer*. O factor de decimação corresponde a um valor inteiro, maior do que 0, tal que dividido pela frequência de amostragem real se obtém a frequência de amostragem efectiva (frequência de amostragem reduzida); esta técnica é conhecida como sub-amostragem. A Figura 27 mostra o diagrama de decimação com factor de M .

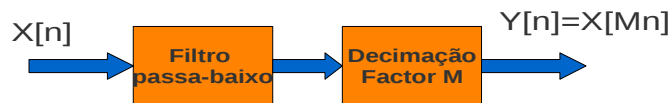


Figura 26: Diagrama de decimação

Para que a sub-amostragem seja correcta é necessário reduzir a largura de banda do sinal usando um filtro discreto passa-baixo. A decimação pode ser entendida como uma compressão no tempo tal que provoca expansão na frequência. Dado que a máxima frequência a ser gerada pela guitarra é de 1174 Hz, como apresentada na Tabela 1, a redução da largura de banda tem que respeitar a ritmo de Nyquist de modo a não cortar esta frequência. Isto significa que é necessário implementar um filtro digital passa-baixo cuja frequência de corte é superior a 1174 Hz.

Para a selecção do filtro passa-baixo, foi feita uma pesquisa por filtros que tivessem uma resposta na frequência de corte com maior precisão para uma ordem menor. A precisão na frequência de corte é importante para que depois da decimação não exista *aliasing* [27] no sinal, situação que iria dificultar a detecção de frequências.

Para esta análise, contribuiu a consulta do site de Tony Fisher [30], que desenvolveu uma página web interactiva para desenho de filtros digitais. Mediante a utilização da página do *mkfilter* [31], desenvolvido por Tony Fisher, foi possível analisar a resposta em frequência de filtros passa-baixo; Bessel [32], Butterworth [32] e Chebyshev [32]. No site com os parâmetros introduzidos, produz como saída coeficientes do filtro e a resposta em frequência. Os testes foram efectuados considerando os valores apresentados na Tabela 6.

Parâmetro	Valor
Frequência de amostragem	44100 Hz
Frequência de corte	2000 Hz
Ordem do filtro	4 e 8
Ripple	-0,01 dB (só para o tipo Chebyshev)

Tabela 6: Parâmetros utilizados para analisar a resposta em frequência dos filtros passa-baixo

Dos 3 filtros, o escolhido foi o de Chebyshev, pelas características da resposta em frequência. Podemos comparar nas Figuras 27, 28 e 29, para um filtro de ordem 4 as características da resposta em frequência para cada um dos filtros. A Figura 27 apresenta a resposta em frequência do filtro passa-baixo de Bessel.

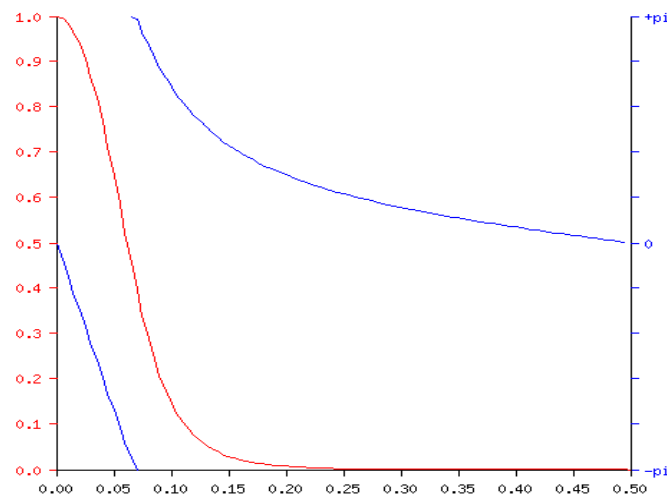


Figura 27: Resposta em frequência do filtro tipo Bessel de ordem 4 (adaptado de [31]). O eixo x representa as frequências em frações da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.

Observamos na Figura 27 que a magnitude na frequência de corte decresce muito lentamente o que significa também que necessita de uma maior ordem para remover efectivamente a frequência desejada. O eixo x representa as frequências em frações da frequência de amostragem, o que significa, por exemplo, 0,5 representa na frequência de Nyquist corresponde a frequência de 22050 Hz. Na Figura 29 apresenta a resposta em frequência do passa-baixo de Butterworth.

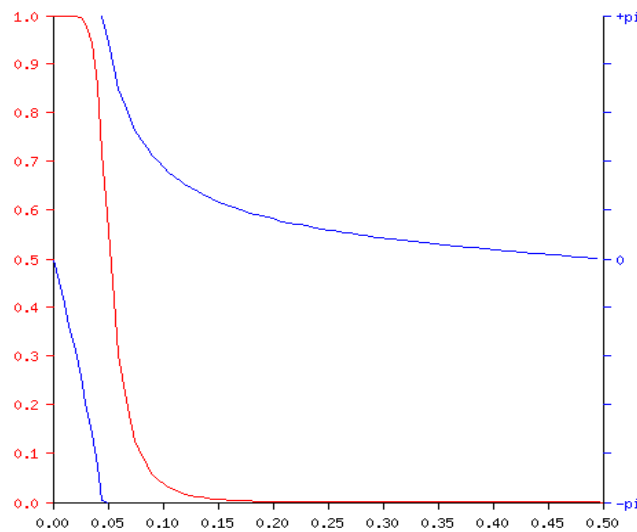


Figura 28: Resposta em frequência do filtro de Butterworth de ordem 4 (adaptado de [31]). O eixo x representa as frequências em frações da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.

Podemos observar comparativamente com o filtro de Bessel, que o de Butterworth, na Figura 28, possui um declive mais acentuado na frequência de corte. A Figura 29 apresenta a resposta em frequência do filtro de Chebyshev.

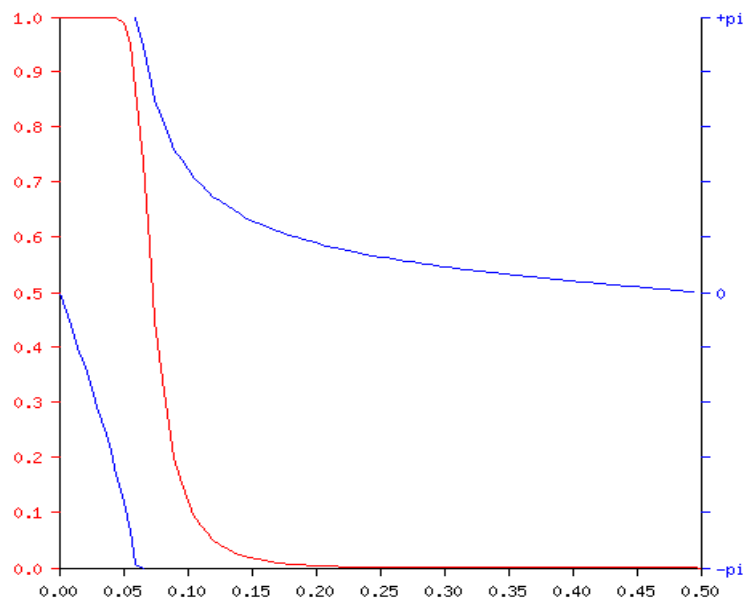


Figura 29: Resposta em frequência do filtro de Chebyshev de ordem 4 (adaptado de [31]). O eixo x representa as frequências em frações da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.

Com o filtro de Chebyshev verifica-se uma banda de transição estreita frequência de corte e o decremento da magnitude é maior que no filtro de Butterworth. Em geral podemos observar que para ordem 4, o filtro de Chebyshev apresenta melhor resultado na remoção da

frequência, mas em nenhum dos casos há uma redução mais próxima de zero da frequência de corte.

No decorrer de várias análises, a conclusão é de que um filtro de ordem 8 seria o suficiente para obter a resposta em frequência desejável para o problema. As Figuras 30, 31 e 32, apresentam as respostas em frequência dos 3 filtros, utilizando ordem 8. A Figura 30 apresenta a resposta em frequência do filtro de Bessel.

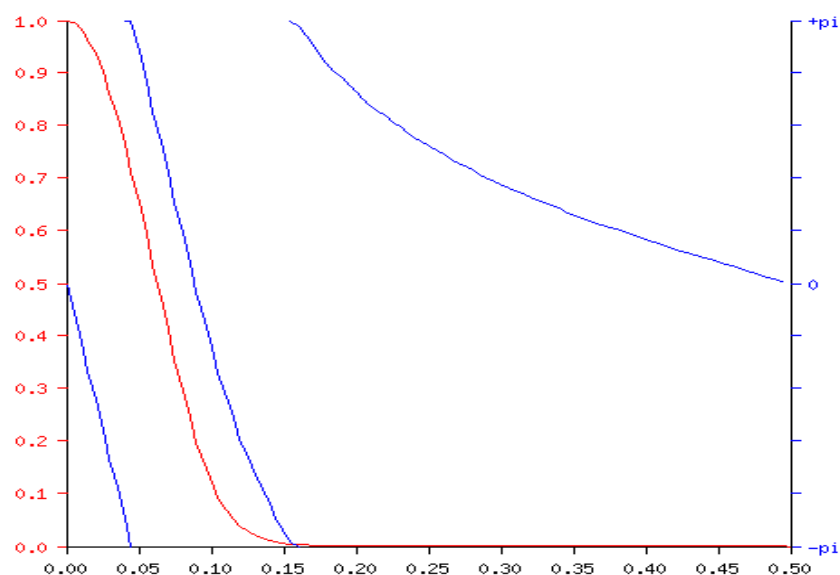


Figura 30: Resposta em frequência do filtro de Bessel de ordem 8 (adaptado de [31]). O eixo x representa as frequências em frações da frequência de amostragem, O eixo y a vermelho representa a magnitude linear e normalizada da resposta do filtro, e y a azul a fase.

A Figura 31 apresenta a resposta em frequência do filtro de Butterworth.

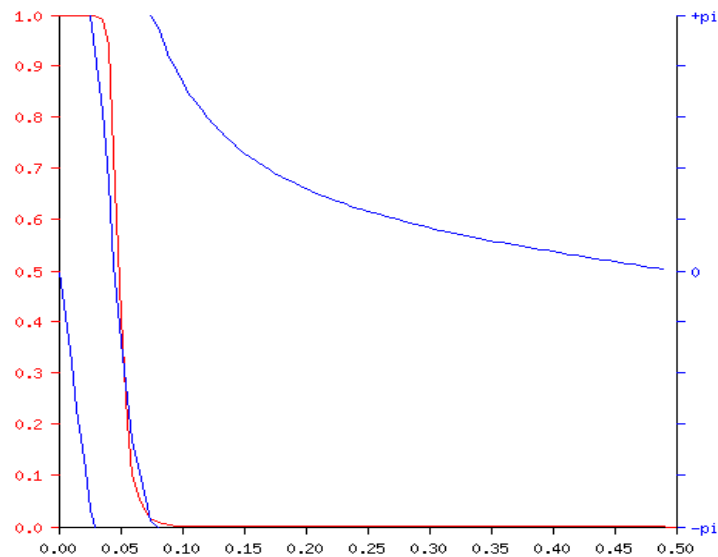


Figura 31: Resposta em frequência do filtro de Butterworth de ordem 8 (adaptado de [31]). O eixo x representa as frequências em frações da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.

Verificamos que comparativamente com o filtro de Bessel, o de Butterworth tem melhor atenuação na banda de corte. A Figura 32 apresenta a resposta em frequência do filtro de Chebyshev.

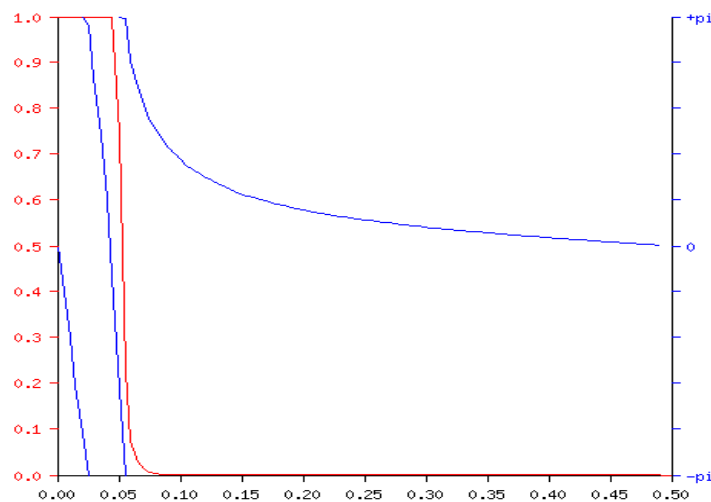


Figura 32: Resposta em frequência do filtro de Chebyshev de ordem 8 (adaptado de [31]). O eixo x representa as frequências em frações da frequência de amostragem, O eixo y a vermelho representa a magnitude e normalizada da resposta do filtro, e y a azul a fase.

Fazendo uma comparação com os gráficos anteriores, onde é apresentada a resposta em frequência para os filtros de ordem 4, Figuras 27, 28 e 29; os filtros de ordem 8 possuem uma banda de transição mais estreita na frequência de corte. Comparando os 3 filtros de ordem 8, podemos observar que o filtro de Chebyshev decremente a sua magnitude mais rápido que o filtro de Butterworth, ou seja, o filtro de Chebyshev tem um declive mais acentuado, tornando o filtro mais adaptado para a solução. A vantagem de ter um declive

mais acentuado na banda de corte, está relacionada com a proximidade a um filtro passa-baixo ideal, e de ser necessário menos iterações ou ciclos para atenuar a amplitude a partir da frequência de corte. Dado que se optou pelo filtro de Chebyshev é necessário estudar mais as características deste filtro para a implementação da solução a ser desenvolvida.

3.5.1 Filtro passa-baixo de Chebyshev

O filtro de Chebyshev é uma estratégia matemática para obter uma atenuação mais rápida na nas frequências de corte, permitindo ter um ganho superior nas frequências da banda de passagem, enquanto reduz o ganho na banda de corte permitindo ter um ganho na frequência de banda de passagem que é chamado *ripple*. Este filtro minimizam o erro entre as características de um filtro ideal e o actual em relação à banda de corte, diferenciando somente no ganho extra nas frequências de passagem. A Figura 33 [33] mostra a resposta em frequência de um filtro passa-baixo do tipo I, de Chebyshev.

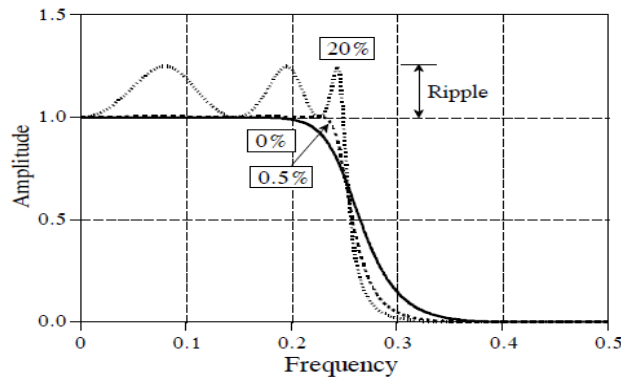


Figura 33: Resposta em frequência do filtro de Chebyshev com ripple de 20% (adaptado de [33])

Na Figura 33 quando o valor do *ripple* é de 0%, que é chamado máximo o filtro tem as características de um filtro de Butterworth, por este motivo o filtro de Chebyshev é uma generalização do filtro de Butterworth. Considerando um *ripple* de 0,5%, observa-se que o ganho é tão pequeno que não é possível ser observado no gráfico da Figura 33. A principal vantagem em relação ao Butterworth está na atenuação mais íngreme na banda de corte, o que significa que o corte da frequência é mais rápido do que o de Butterworth, o que significa que o filtro de Chebyshev necessita de menor ordem para se aproximar às características de um filtro ideal. Existem 2 tipos de filtros de Chebyshev: I e II [35]. A principal diferença está no tipo II ser inversa do filtro do tipo I, como exemplificado nas Figuras 34 e 35, respectivamente.

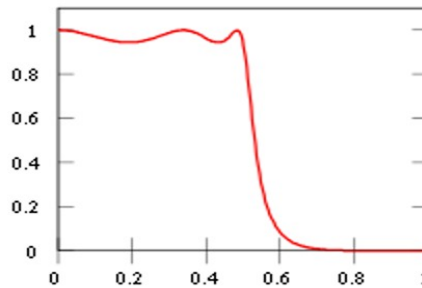


Figura 34: Resposta em frequência do filtro de Chebyshev tipo I (adaptado de [34])

Na Figura 34 observamos o *ripple* na banda de passagem e uma rápida atenuação na banda de corte, no tipo II o *ripple* está na banda de corte e atenuação não é tão acentuada, como apresenta a Figura 35 [34].

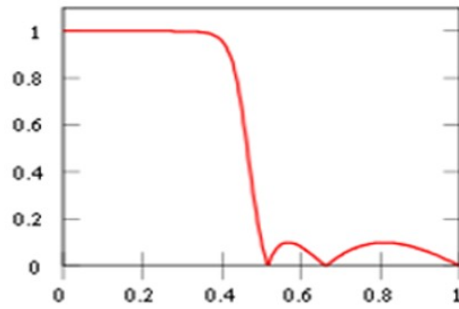


Figura 35: Resposta em frequência de filtro de Chebyshev tipo II (Adaptado de [34])

Para o problema a resolver será utilizado o tipo I, porque o tipo II não atenua tão rápido na banda de corte. O ganho constante na banda de passagem não é um requisito importante. Desde que a frequência das notas não seja removida ou muito atenuada, é possível detectar a frequência.

3.5.1.1 Implementação do Filtro

O filtro será implementado com uma estrutura *Infinite Impulse Response* (IIR), dado as características do filtro de Chebyshev apresentar pólos. Caracteriza-se como filtro IIR um filtro digital cuja resposta impulsional é diferente de zero num tempo infinito. Também é caracterizado por não só por atenuar frequências mas também por adicionar ganho as frequências que se pretende deixar passar. Este pode ser descrito em termos de equação às diferenças como

$$y[n] = \sum_{k=0}^M b_k x[n-k] + \sum_{k=1}^N a_k y[n-k], \quad (18)$$

onde $\max(M,N)$ é ordem, a e b os coeficientes, representando os pólos e zeros respectivamente e $x[n]$ o sinal de entrada. A equação às diferenças (18) define como o sinal de saída está relacionado com o sinal de entrada, e pode ser representado graficamente em forma de diagrama, usando a forma directa II como apresenta a Figura 36 [36].

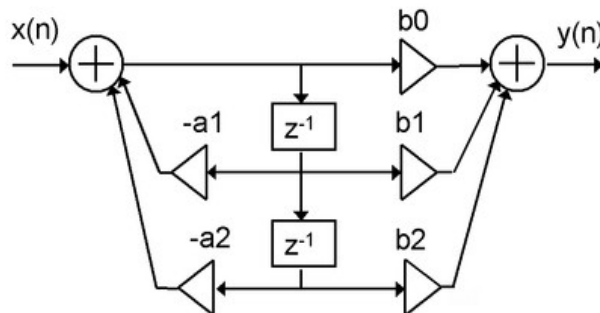


Figura 36: Diagrama de filtro IIR na forma directa II (Adaptado de [36])

Na forma directa II é uma estrutura que permite a implementação de filtro IIR de forma otimizada, consumindo menores recursos a nível computacional. Também podemos representar a equação (18) na forma de transformada Z, e obtemos

$$H(z) = \frac{b_0 \prod_{k=1}^M (1 - b_k z^{-1})}{\prod_{k=1}^N (1 - a_k z^{-1})}. \quad (19)$$

Na equação (19) podemos ver que a função de transferência de um filtro IIR é uma função racional em z^{-1} . Dado que o filtro de Chebyshev é originalmente um filtro analógico e para sinais contínuos, é possível transformar num filtro digital mediante aplicação de uma transformação bilinear. A transformação bilinear é um método para transformar um sistema de tempo contínuo noutro no tempo discreto e vice-versa. Baseia-se em mapear os valores presentes num sistema em outro, usando mudança de variável. Para o filtro, será necessário mapear o pólos encontrados no planos tal como apresentado em (45) anexo B, para o plano z na forma da equação (19).

3.6 Técnicas de janelas

As técnicas de janelas de um sinal são conhecidos por poderem melhorar as características espectrais do sinal, e desta forma melhorar a análise espectral usando diferentes algoritmos tais como os mencionados neste capítulo. A técnica de janela consiste em aplicar e multiplicar as amostras do sinal por uma função de janela. Existem várias funções de janelas, como por exemplo: Rectangular [37], Hanning [37], Hamming [37], Blackman [37] entre outros.

A função de Hamming é dada por

$$W[n] = 0.54 - 0.46 * \cos\left(2\pi \frac{n}{N}\right), \quad n=0,1,2,\dots,N-1. \quad (20)$$

Depois de calculados os coeficientes da janela, esta pode ser multiplicada pelas amostras de entrada, e ao conjunto de amostras resultante é aplicado o algoritmo de estimação de frequência.

3.7 Técnica de estimação da duração da nota

Nesta secção analisa-se a estimação da duração da nota. Calculando a duração de cada nota será possível, por exemplo elaborar uma partitura e fazer a conversão em MIDI das notas recebidas.

É necessário criar um metrónomo para definir a velocidade da música, um quantizador, para quantizar a notas por cada batida do metrónomo. A quantização das notas corresponde à contagem do número de notas que ocorrem em cada batida de metrónomo. Definindo a quantização das notas podemos construir uma pauta musical de diferentes formas, porque definimos quais são as bases de figuras musicais que serão apresentadas na pauta musical

por cada batida, mediante a duração de cada nota. Este facto depende da relação que as figuras das notas musicais têm entre si.

Para estimar a nota é necessário contar a duração de cada amostra de sinal que é recebido, mas também é necessário estimar as pausas, ou seja, a duração do silêncio. Os seguintes parâmetros são necessário ter em conta quando pretendemos calcular o duração da nota: BPM, frequência de amostragem, compasso escolhido pelo utilizador, quantização das notas e o número de amostras recolhidas do sinal de entrada.

Seja N o número de amostras recolhidas à frequência de amostragem, F_s podemos dizer que

$$SampleInterval = N * \frac{1}{F_s} \text{ [ms]} \quad (21)$$

representa o intervalo de tempo que se recolhe um conjunto de N amostras em milissegundos. Seja BPM o número de batidas por minuto, em segundos, obtemos

$$BeatInterval = 60 * \frac{1000}{BPM} \text{ [ms]}, \quad (22)$$

que representam em milissegundos o intervalo de tempo entre cada batida de metrónomo. Com base no compasso musical seleccionado, calcula-se o tempo total de um compasso na partitura usando (23), dada por

$$TimeDivision = Compass * BeatInterval \text{ [ms]}. \quad (23)$$

Com o valor de quantização de notas, podemos calcular o intervalo mínimo de uma nota ou pausa

$$QuantizationInterval = \frac{BeatInterval}{Quantization} \text{ [ms]}. \quad (24)$$

Com as equações (21), (22), (23) e (24), podemos estimar a duração de cada nota ou pausa musical utilizando duas estratégias. Uma solução seria utilizar uma janela deslizante que calcula a energia na *frame* áudio. Quando esta energia inicia e diminui até um limiar mínimo é considerado que o sinal terminou; quando a energia é maior que um limiar máximo é considerado como início. A Figura 37 ilustra como a janela deslizante passa pelo *buffer* de amostras, calculando a energia.

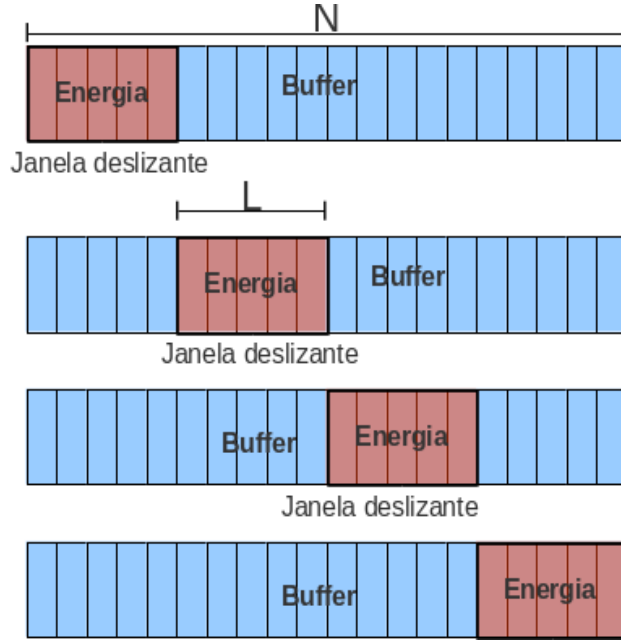


Figura 37: Sequência da janela deslizante para o cálculo da duração do sinal

Na Figura 37, N representa o tamanho do *buffer* e L da janela deslizante. Dado que a frequência de amostragem é fixa, o índice do início do sinal e o índice do fim da nota vão fornecer a distância entre o tempo final e o tempo inicial. Esta distância multiplicada por *SampleInterval* (21) e determina a duração da nota tocada na *frame* e é dada como

$$Duration = SampleInterval * (EndIndex - StartIndex) \text{ [ms]}, \quad (25)$$

onde *EndIndex* e *StartIndex* correspondem ao índice inicial e final, respectivamente. No entanto esta solução dada a frequência de amostragem e o número de amostras recolhidas necessita de uma dimensão de *buffer* demasiado elevada. A seguir é apresentado um exemplo para demonstrar a problemática desta solução.

Exemplo: Dado um compasso de 4, o metrónomo a 100 BPM, quantização igual a 4 (significa que numa batida de metrónomo podem existir 4 notas), o número de amostras é igual a 1024 e a frequência de amostragem é de 44100 Hz, vamos calcular o tamanho do *buffer* de amostras e o tamanho da janela deslizante.

Calcula-se o *SampleInterval* usando (21)

$$SampleInterval = 1024 * \frac{1}{44100} = 23,22 \text{ [ms]}$$

Calcula-se o intervalo de cada batida usando (22)

$$BeatInterval = 60 * \frac{1000}{100} = 600 \text{ [ms]},$$

calcula-se o tempo total de cada divisão com a expressão,

$$TimeDivision = 4 * 600 = 2400 \text{ [ms]}.$$

Com o valor da quantização calculamos o intervalo mínimo de cada 4 notas numa batida de 100 BPM

$$QuantizationInterval = \frac{600}{4} = 150 \text{ [ms]}.$$

Assumindo que se pretende determinar a menor duração de nota, calculamos o tamanho do *buffer* necessário para determinar a presença de uma nota de 150 ms

$$BufferSize = \frac{QuantizationInterval * N}{SampleInterval} = \frac{150 * 1024}{23,22} = 6614,98.$$

Significa que para estimarmos um sinal de 150 ms de duração nas condições especificadas neste exemplo precisaríamos de ter um *buffer* de 6615 pontos, que iria ocupar muitos recursos computacionais. É necessário realçar que o aumento do número de batidas do metrónomo diminui o tamanho necessário para detectar 4 notas numa batida, e o processo contrário aumenta o tamanho do *buffer*. Os valores de quantização também influenciam o tamanho do *buffer*: a diminuição deste aumenta o tamanho do *buffer* e vice-versa.

Uma solução menos dispendiosa a nível de processamento seria contar simplesmente quanto tempo uma dada frequência se mantém presente. Dado que o intervalo das 1024 amostras ser fixo, basta contar o número de vezes seguidas que a nota é detectada, ou seja, quando uma dada frequência está presente, o valor da sua duração é incrementado até que a frequência mude de valor. Em seguida, podemos estimar a duração da nota usando a relação entre *QuantizationInterval* e *SampleInterval*, dada por

$$fraction = \frac{QuantizationInterval}{SampleInterval} = \frac{150 \text{ ms}}{23,22 \text{ ms}} = 6,45, \quad (26)$$

que representa o número de vezes que temos que receber uma frequência para determinar que a sua duração foi de 150 ms. Desta forma, contando o número de vezes consecutivas que a frequência é detectada, podemos determinar a sua duração. A mesma solução é válida para a duração da pausa.

3.8 Resumo

Em resumo como algoritmo de estimação de frequência optou-se pela FFT. Para resolver o problema do número de pontos é necessário aplicar decimação de maneira a reduzir a frequência de amostragem. Também foi decidido, que para a aplicação da decimação é necessário um filtro passa-baixo, e optou-se pela utilização do filtro de Chebyshev tipo I.

Dado que reduzimos a frequência de amostragem por um factor de decimação, significa que podemos então definir qual é o número de pontos necessários para detectar as frequências. Como a frequência máxima a ser processada é de 1174 Hz, como apresenta a Tabela 1, significa que podemos encontrar o factor de decimação que reduza a frequência até

amostragem até $2 \times 1174 \text{ Hz} = 2348 \text{ Hz}$, respeitando assim a frequência de Nyquist. Neste sentido significa que para uma frequência de amostragem igual 44100 Hz , o factor de decimação máximo para respeitar a frequência de Nyquist é dado por,

$$\text{factor decimação} = \frac{f_s}{f_n} = \frac{44100}{2348} = 18,78 \quad (44),$$

onde f_s é a frequência de amostragem e f_n a frequência de Nyquist. Calculando a resolução no tempo e na frequência, obtemos os valores apresentados na Tabela 7, considerando o factor de decimação igual a 18 e a frequência de amostragem 44100 Hz a taxa efectiva de amostragem será de 2450 Hz .

Numero de pontos	Resolução em frequência (fs/N) [KHz]	Resolução no tempo (N/fs) [ms]
1024	2,39	0,4
512	4,78	0,2
256	9,57	0,1
64	38,28	0,02

Tabela 7: Número de pontos da FFT, resolução no tempo e frequência

Verifica-se na Tabela 7, que com a decimação é possível obtermos uma resolução mais baixa com um número de pontos inferiores ao apresentados na Tabela 3, e que a partir de 512 a resolução em frequência aumenta, diminuindo a precisão. Com estes resultados o número de pontos a ser utilizado para é 512. A Tabela 8 resume todos os parâmetros a serem utilizados na estimação da frequência.

Parâmetros	Descrição / Valor
Algoritmo de estimação de frequência	FFT
FFT número de pontos	512
Técnica de Janela	Hamming
Número de pontos da janela	512
Decimação	No tempo
Factor de decimação máximo	18
Filtro passa-baixo	Chebyshev Tipo I
Número de pólos	8
Frequência de corte	2000 Hz
Ripple	0,5 dB
Largura de banda da guitarra eléctrica	82 à 1174 Hz

Tabela 8: Resumo dos parâmetros utilizados para estimação da frequência

Para melhorar a qualidade do espectro do sinal, optou-se por aplicar uma janela de Hamming antes de se calcular a FFT, suavizando assim o sinal. A opção por utilizar a frequência de amostragem de 44100 Hz resume-se pelo facto de ser a frequência de amostragem mais utilizada em áudio profissional, principalmente para gravação do sinal

vindo da guitarra eléctrica. Mas este valor não é fixo, depende exclusivamente da opção do utilizador.

No Capítulo 6 são apresentados resultados referentes aos testes efectuados com estes parâmetros. Analisa-se também os valores ideais dos parâmetros que minimizam o erro de detecção de frequência.

4 Solução proposta e implementação

Neste capítulo é estudada e analisada a solução para o problema apresentado no capítulo 2. Apresenta-se pormenores sobre a solução implementada e detalhes de implementação, bem como as ferramentas utilizadas para o desenvolvimento da mesma.

4.1 Critérios de desenho da solução

A escolha da tecnologia a ser utilizada no desenvolvimento do projecto foi baseada nas seguintes considerações:

- Suporte multi-plataforma para os sistemas operativos , Linux e Windows.
- Maior reutilização possível do código.
- Uso de programação concorrente, programação paralela e processamento digital de sinal para detecção de frequência.
- Ambiente gráfico para a aplicação, ilustrando as funcionalidades implementadas.

Para programação concorrente a primeira decisão foi de utilização da API Pthread [38], porque pode ser aplicada em qualquer um dos sistemas operativos alvo, com a linguagem C++. No caso específico do Windows, será utilizada a biblioteca Pthreads-Win32 [38]. No decorrer do desenvolvimento, com alguma pesquisa, foi encontrada a biblioteca Boost C++ [39]; Boost é um conjunto de bibliotecas C++ com o objectivo de estender as funcionalidades da linguagem C++, com suporte em vários sistemas operativos.

Boost possui uma biblioteca de programação concorrente que se chama Boost Thread [40]. Esta biblioteca é utilizada no desenvolvimento da aplicação em detrimento da API PThread. A grande vantagem de se utilizar esta biblioteca, reside no desenvolvimento de código C++ genérico e portátil para qualquer outra plataforma, sem que seja necessário a inclusão no projecto de API externas. A biblioteca Boost Thread, disponibiliza para além de métodos para criação de *thread*, objectos de sincronismo como *mutex*, semáforos, variáveis condicionais e outros objectos de sincronismo como *barries* e *future*. Toda a programação concorrente é desenvolvida fazendo uso da biblioteca Boost Thread.

Outra vantagem da utilização da biblioteca Boost no projecto é a biblioteca Signal e Slots, Boost Signals [41]. A biblioteca Boost Signal implementa funcionalidades semelhantes a eventos e delegate em C#. Os *signals* representam *callbacks* com múltiplos alvos, e os *slots* são *callbacks* receptores os quais o *signals* estão ligados e que são executados quando um sinal é emitido. A biblioteca Boost Signal2 será utilizada no projecto para notificações, como por exemplo para iniciar a captura do sinal áudio ou outras notificações entre as diferentes camadas da aplicação.

Para o projecto será utilizada a segunda versão da biblioteca Boost Signal, designada por Boost Signals2 [42]. A diferença entre as duas versões é que a versão 2 suporta *multi-threading*.

Para programação paralela, a escolha é a utilização de OpenMP [43] para a linguagem C/C++. A opção de utilização do OpenMP resulta de 2 factores; o paralelismo ser introduzido por meio de directivas incluídas no código, o que não exige grandes mudanças neste; o facto de

estarmos numa arquitectura com memória partilhada. Na Figura 38 [44] é apresentada a arquitectura de um sistema multi-processor ou *multicore*.

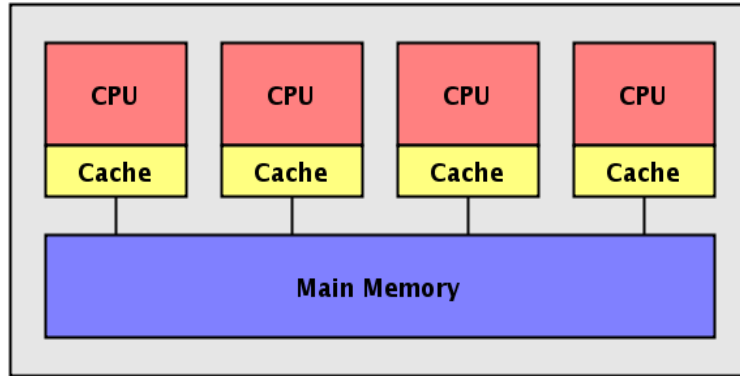


Figura 38: Exemplo de uma arquitectura multicore ou multi-processor (Adaptado de [44])

O OpenMP é utilizado para paralelismo funcional e permite tirar partido da tecnologia *multicore*. Para paralelismo de dados pretende-se utilizar Assembler com instruções Single Instruction Multiple Data (SIMD) para caso de alguma optimização a nível de operações com dados.

Para a aplicação de técnicas de processamento de sinal digital, é utilizada a biblioteca FFTW [45] com a linguagem C/C++. Esta biblioteca suporta diferentes sistemas operativos sem que seja necessário alterar o código. Outra característica desta biblioteca é estar desenvolvida com suporte de SSE/SSE2/3DNow; estas instruções pertencem ao conjunto de instruções chamadas SIMD. As instruções SIMD permitem num único ciclo de CPU efectuar operações em dados armazenados de forma vectorial. Esse conjunto de instruções permite manipular vários dados simultaneamente, o que divide o tempo de operação, dos n vectores, por ciclo de CPU. A Figura 39 [46] exemplifica a comparação entre uma operação escalar e uma operação com SIMD.

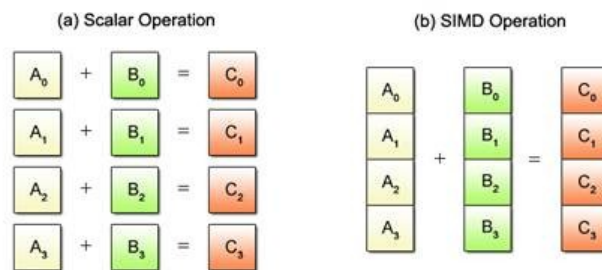


Figura 39: Comparação entre operações escalares e operações SIMD (adaptado de [46])

Com as instruções SIMD o cálculo da FFT [10] pode ser efectuado de forma mais eficiente em computadores que suportem estas instruções, porque é possível operar vários dados em simultâneo em cada ciclo de CPU.

Para suporte a áudio será utilizada a biblioteca RTAudio [47] em C/C++, que suporta a gravação e reprodução de áudio em tempo real nos sistemas operativos Linux, Windows e MacOSX. Esta biblioteca simplifica o desenvolvimento antes iniciado para suportar os

vários *drivers* de áudio nos diferentes sistemas operativos. A biblioteca tem a capacidade de enumerar os dispositivos áudio, multi-canal e conexão dinâmica entre dispositivos áudio.

Para gravação de ficheiros áudio, é utilizada a biblioteca libsndfile [48]. Esta biblioteca permite a leitura e gravação de ficheiros áudio em diversos formatos. Suporta a definição da frequência de amostragem, número de canais e está disponível para qualquer sistema operativo.

4.2 Arquitectura

A aplicação tem uma arquitectura cliente-servidor, onde as duas componentes principais, o *GSTTransporter* e a interface gráfica, que são respectivamente servidor e cliente. O *GSTTransporter* ou simplesmente *Transporter* corresponde a um executável que processa todo o sinal de entrada produzido pela guitarra eléctrica, e tem como saída as notas musicais que foram tocadas pelo guitarrista, e um ficheiro áudio contendo a gravação de todo o sinal produzido pela guitarra eléctrica. A Figura 40 ilustra a arquitectura.

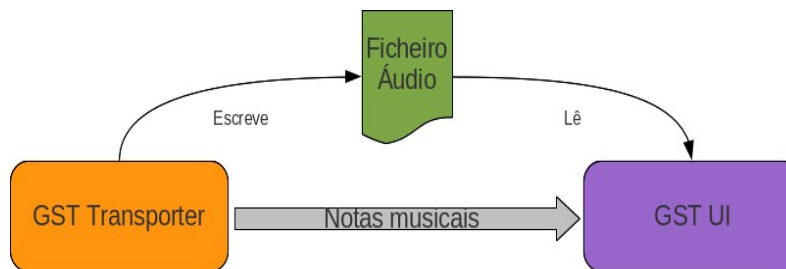


Figura 40: Arquitectura da aplicação

A interface gráfica é por onde o utilizador fará a interacção com a aplicação, podendo visualizar as notas produzidas pela guitarra, e é alimentada pelas notas debitadas pelo *GSTTransporter*.

A interface de captação de áudio será responsável pela aquisição de áudio na plataforma, e a sua implementação será dependente do sistema operativo. Esta interface permite a interacção com a placa de som ou outro servidor de áudio (Audio Stream Input Output -ASIO no Windows ou Jack Audio Connection Kit-JACK no caso do Linux) para passar o sinal áudio à camada de processamento de áudio. Esta interface também irá definir a frequência de amostragem para aquisição do sinal. Na Figura 41, apresentam-se as camadas da aplicação.

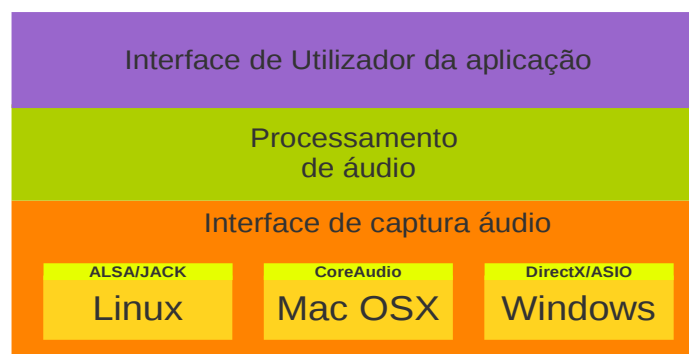


Figura 41: Camadas da aplicação

A camada de processamento de áudio realizará detecção de frequência e estimação da duração do sinal áudio vindo da interface de captação. Cada janela do sinal áudio que for captado, será processada numa *thread* da *thread pool*, dando a possibilidade de processar a próxima amostra áudio.

Como a camada de processamento áudio somente detecta a frequência e calcula a duração do sinal, estes dados são enviados para serem interpretados pelas aplicações que se ligarem ao *Transporter*. É da responsabilidade da interface gráfica dar a interpretação correcta da informação enviada pelo *Transporter*, e isto significa que o mapeamento de frequências para notas está fora de âmbito do *Transporter*.

A interface de utilizador, irá permitir a interacção do guitarrista com a aplicação e receberá as informações das notas tocadas da camada de transformação para apresentar ao utilizador. Na interface de utilizador é necessário prever todas as sequências de acções que o utilizador possa efectuar. Parte do processamento áudio foi desenvolvido no trabalho preliminar, como a detecção de frequência e mapeamento para uma nota musical.

As várias camadas serão agrupadas, serão constituídos componentes com as diferentes camadas. O objectivo é ter em separado a componente de processamento das notas e a de processamento de sinal captado pela guitarra eléctrica. A Figura 42 resume as bibliotecas e suas aplicações.



Figura 42: Diagrama de blocos resumindo as bibliotecas utilizadas e sua aplicação

5 Aspectos de implementação e desenvolvimento

Neste Capítulo são abordados aspectos relativos ao desenvolvimento da aplicação. São descritas as várias classes que compõem a aplicação e as suas características. A descrição de cada componente está dividida pelas diferentes camadas. Existem 2 componentes principais da aplicação: o *GSTTransporter* e a interface gráfica. O *GSTTransporter* tem como finalidade a captação, processamento e transformação do sinal da guitarra eléctrica. A interface gráfica tem como funcionalidade a apresentação das notas tocadas, como a possibilidade de correcção feita pelo guitarrista.

5.1 GST Transporter

O *GSTTransporter* implementa todas as funcionalidades relativas à sequência de processamento das amostras recebidas da guitarra até ao resultado final que são as notas tocadas pelo guitarrista. Como resultado obtemos um ficheiro executável. Ao executável são passados os seguintes parâmetros, apresentados na Tabela 9.

Parâmetros	Descrição
SampleRate	Valor para a frequência de amostragem
NumBuffer	Número de buffer (double buffering, triple)
BufferSize	Tamanho do buffer do servidor áudio
NumThreads	Número de threads que serão usados na <i>pool</i> de <i>threads</i>
AudioFormat	Formato áudio que se deseja gravar o ficheiro
AudioFileName	Nome do ficheiro áudio
AudioDriver	<i>Driver</i> que será utilizado para a captura
AudioCDevice	Placa de som utilizada para captura
AudioPDevice	Placa de som utilizada para playback
SystemInfo	Retorna informação relativa aos drivers e placas de som encontradas no sistema onde está a executar.
Log	Define se aplicação faz log ou não

Tabela 9: Parâmetros passados ao executável do Transporter

Os parâmetros configuram a execução do *GSTTransporter*, para depois poder iniciar todo o processo de captura. Este componente disponibiliza uma interface para comunicação em tempo real entre a interface gráfica de onde serão dadas as ordens para começar a captura e processamento dos sinais vindos da guitarra eléctrica.

Os parâmetros permitem iniciar o servidor áudio com a escolha da placa de som que será utilizada para captação e para reprodução, no caso de existir mais do que uma placa de som no computador do guitarrista. O *driver* seleccionado serve para escolher qual o software que será utilizado sobre as placas de som.

Alguns parâmetros são partilhados pelas diferentes camadas que compõem o *GSTTransporter*. Outros são utilizados apenas por camadas específicas tais como *AudioFileIO*, para definir o nome do ficheiro áudio de saída, por exemplo.

Para analisar os parâmetros passados ao executável, é utilizada a biblioteca Boost `program_options` [49]. Esta biblioteca permite obter opções do programa quer sejam passadas como parâmetro ou por ficheiro de configuração. A Figura 43 ilustra a sequência de processamento de sinal efectuada pelo *Transporter*.

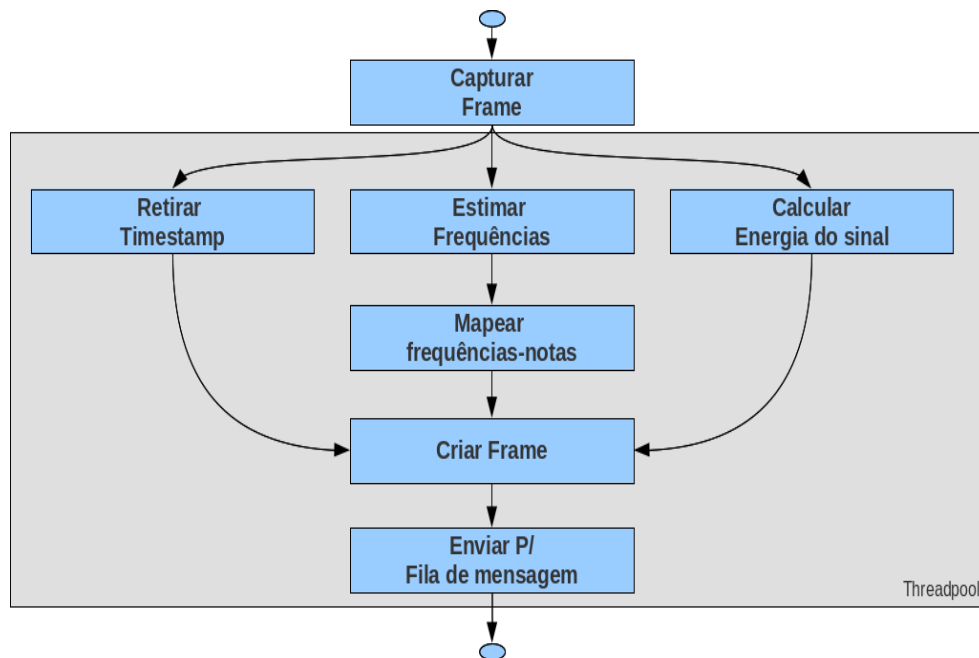


Figura 43: Sequência de processamento do *Transporter*

O processamento é feito numa *threadpool*, através de diferentes *thread* a processar os dados sem que o *Transporter* perca amostras do sinal seguinte. Como a Figura 43 ilustra, a *frame* contendo o sinal da guitarra é captada, e é registado o tempo que a *frame* foi captada (*timestamp*) e calculada a duração do sinal e é iniciada a estimação da frequência. Quando estes 3 processos terminam é criado um objecto que contém 7 dados: *timestamp*, índice, 6 frequências, energia da frame, o pico máximo e mínimo, e a duração. Depois de criada a nota, esta é posta na fila de mensagem, que é acessível para outras aplicações poderem receber. A limitação de 6 notas musicais, está relacionada com o facto da guitarra ter 6 cordas, e premindo os trastos o máximo de notas que poderão ser tocadas em simultâneo são 6.

O processamento de cada *frame* é executado em *threads* diferentes e neste sentido não há garantia de qual será o processamento que terminará primeiro. Assim, o *timestamp* regista o valor do tempo no momento em que a *frame* do sinal é adquirida, para determinar a ordem certa de cada *frame*. Desta forma, as notas recebidas pelos clientes serão sempre interpretadas na ordem certa.

A estimação das frequências do sinal possui a sequência ilustrada na Figura 44. Esta sequência é executada pelas *threads* pertencentes à *threadpool* sempre que receberem uma amostra nova do sinal da guitarra eléctrica. A *threadpool* foi desenvolvida usando a biblioteca Boost `threadpool` [50], que permite a sua utilização de forma independente do sistema operativo.

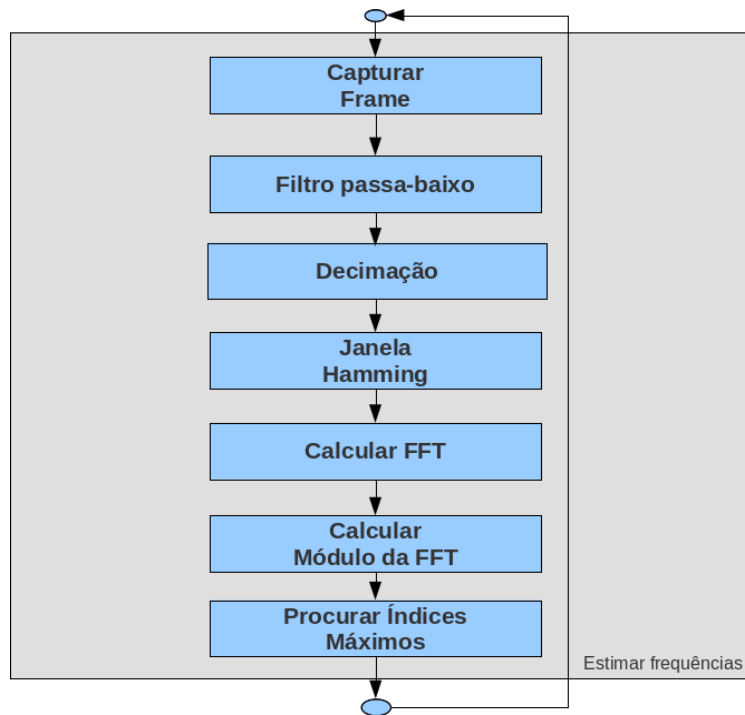


Figura 44: Sequência para a estimação de frequências

Quando chega uma *frame* contendo o sinal da guitarra, esta é passada por um filtro passa-banda. Este filtro irá retirar todas as frequências que não pertencem à largura de frequência da guitarra. Em seguida é feita a decimação do sinal baseado num factor de decimação escolhido. Aplica-se ao sinal decimado uma janela de Hamming e posteriormente efectua-se o cálculo da FFT.

Com a potência do espectro, é feita uma pesquisa pelos máximos. Dado que podemos ter várias notas tocadas numa *frame*, foi definido um valor limiar de potência a qual se considera máximo. Assim, somente se procuram os índices cujo valor ultrapasse o limiar. Com os índices destes valores máximos, calcula-se a frequência correspondente e a nota musical.

As notas são obtidas através da aproximação entre duas frequências com as respectivas notas. É dividido o valor de frequência calculada na estimação pelo intervalo superior e inferior entre duas notas, o valor mais próximo de 1 é assumido como sendo a nota mais próxima associada à frequência.

O *GSTTransporter* devolve as frequências e durações detectadas e não as representações das notas; desta forma conseguimos também garantir que cada cliente possa dar o seguimento que melhor pretende com esta informação: duração e frequência. Um exemplo de aplicação seria a transformação destas notas em MIDI [4].

O *GSTTransporter* é constituído por várias camadas: captura de áudio, processamento de sinal, gravação, e *timer*. Também faz a recepção dos comandos vindos da interface gráfica e executa as operações que foram transmitidas. A comunicação do *GSTTransporter* com outras camadas é feita através de mecanismo de comunicação baseado em sinais, implementado com Boost Signals. Desta forma quando o *GSTTransporter* recebe a ordem de iniciar a captura, envia um sinal para todas as outras camadas para iniciarem também o

processo de captura. Na Tabela 10, estão listados os ficheiros que constituem o *GSTTransporter*.

Classes	Descrição	Ficheiro
CGSTTransporter	Implementa as funcionalidade do <i>Transporter</i> , recepção de comandos e ligação com outras camadas	Transporter.h Transporter.cpp
Frame	Estrutura de uma Frame detectada pelo <i>Transporter</i>	Frame.h
Config ConfigManager	Estrutura com dados de configuração	Config.h
main	Início da aplicação	main.cpp
CLogger	Escrita de log de mensagens do <i>Transporter</i>	Logger.h Logger.cpp
Commands	Enumerado dos comandos do <i>Transporter</i>	Commads.h

Tabela 10: Ficheiros e classes pertencentes ao GSTTransporter

O diagrama UML da classe que implementa o CGSTTransporter é apresentado na Figura 45.

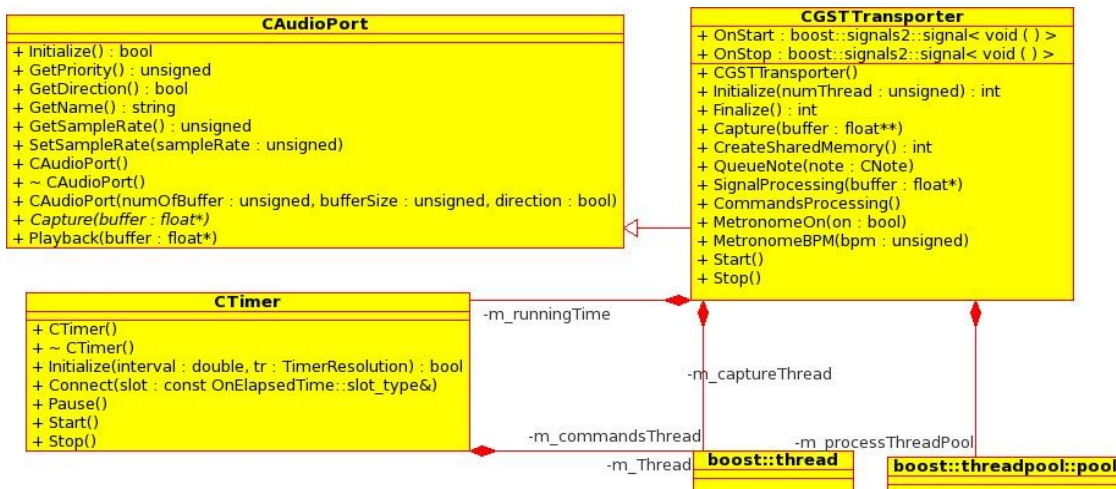


Figura 45: Diagrama UML da classe CGSTTransporter

Na Figura 45 podemos ver o diagrama de classes relacionadas com o *Transporter*. A classe CGSTTransporter deriva de CAudioPort para permitir a esta componente ser adicionada ao servidor áudio. A classe Config contém membros que representam os valores para configurar o *Transporter*, como a frequência de amostragem, número de *buffers* entre outros.

É de salientar que o *Transporter* é totalmente desacoplado da interface gráfica, o que permite exportar o mesmo para outras plataformas tendo somente que se implementar a interface gráfica.

5.1.1 Transporter comandos

O *GSTTransporter* possui uma *thread* responsável por receber comandos enviados pela interface gráfica. Os comandos são guardados numa fila de mensagem implementada

utilizando o Boost message_queue [51]. O Boost message_queue envia e recebe *bytes* na fila de mensagens. Para enviar objectos é necessário que estes sejam serializáveis.

As mensagens estão definidas como sendo do tipo inteiro, e cada valor inteiro passado ao *Transporter* corresponde a um determinado comando. O *GSTTransporter* possui uma *thread* que fica bloqueada na fila de mensagens enquanto esta estiver vazia. Quando a fila recebe uma mensagem a *thread* é libertada e processa o código de mensagem recebida. Na Tabela 11 estão listados os comandos passados ao servidor e os seus códigos.

Comando	Descrição
START	Inicia a captura e processamento das amostras
STOP	Pára a captura e o processamento das amostras
EXIT	Terminar o processo do <i>Transporter</i>
TUNER MODE ON/OFF	Liga e desliga o modo de afinador
RUN BENCHMARK	Calcular os melhores parâmetros para detecção de frequência

Tabela 11: Comandos passados ao GSTTransporter

O comando para o modo afinador permite configurar o *Transporter* com diferentes valores para detectar somente as frequências correspondentes à afinação das 6 cordas da guitarra eléctrica. O comando *RUN BENCHMARK* ordena a execução de uma avaliação dos melhores parâmetros para configurar o *Transporter* de modo a obter menor erro possível na estimação das notas. Como saída produz um ficheiro Extensible Markup Language (XML) [52] que contém os valores das notas a serem estimadas comparativamente com os valores estimados. Também contém informação relativamente ao erro máximo, mínimo e médio da estimação de todas as notas.

A classe CGSTTransporter através de comunicação baseada em sinal faz a activação de funcionalidades noutros componentes tais como o de gravação de áudio, mas certas alterações só podem ser efectuadas quando o *GSTTransporter* não está a ser executado. Esta limitação está relacionado com o facto de existirem parâmetros como a quantização e o BPM, que não podem ser alterados durante o processo em que o *Transporter* está a fazer a estimação da duração da nota, porque poderá devolver dados incorrectos.

5.1.2 Classe CLogger

A classe CLogger é abstracta, sendo responsável por escrever um ficheiros de *log* do *Transporter*. A Figura 46 apresenta o diagrama UML da classe CLogger.

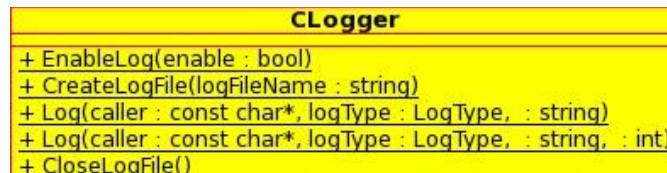


Figura 46: Diagrama UML da classe CLogger

Na Figura 46 observamos os métodos estáticos para escrita do ficheiro *log*, sendo desta forma possível ver as chamadas que estão a decorrer no *Transporter*.

5.1.3 Classe Frame

A classe Frame representa uma *frame* recebida do servidor áudio e processada pelo *Transporter* onde são calculados as componentes do sinal. A Figura 47 apresenta o diagrama UML da classe Frame.

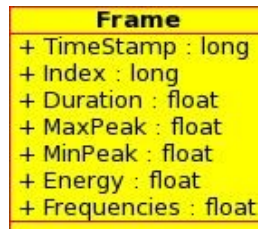


Figura 47: Diagrama UML da classe Frame

Na Figura 47 observamos os membros da classe Frame. O TimeStamp serve para guardar o tempo em que a *frame* foi processada, o Index corresponde o índice da *frame*, Duration corresponde a duração, MaxPeak e MinPeak correspondem ao valores máximo e mínimo existentes na *frame*, respectivamente. O membro Energy corresponde a energia na *frame* e Frequencies é um *array* contendo as frequências detectadas na *frame*.

5.1.4 Classe Config e ConfigManager

A classe Config contém todos os parâmetros de configuração do *Transporter* como membros. A classe ConfigManager tem métodos estáticos para ler e gravar ficheiros de configuração. A Figura 48 apresenta o diagrama UML de ambas as classes.

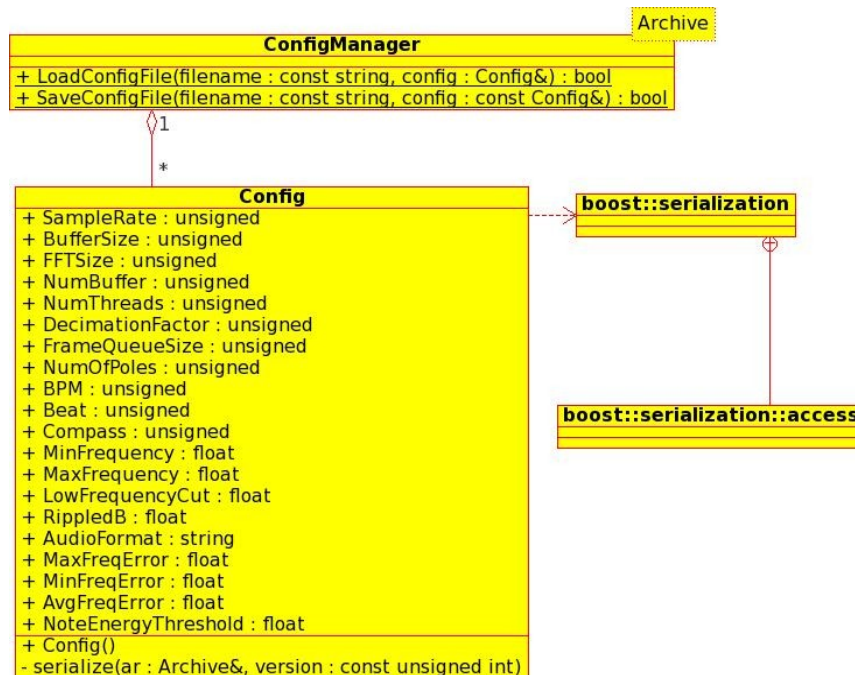


Figura 48: Diagrama UML das classes Config e ConfigManager

Podemos observar na Figura 48 que a classe ConfigManager agrega um objecto do tipo Config, e o Config depende da classe boost::serialization. A dependência da classe Boost

serialization existe para permitir a serialização da classe Config num ficheiro XML. A classe ConfigManager é a responsável pela escrita e leitura do ficheiro mediante a passagem de um objecto do tipo Config.

A classe Config é de grande importância, porque será onde estarão guardados vários parâmetros que servirão também para configurar o processamento de sinal, e pode ser utilizada pela interface gráfica para ler ou escrever novas configurações antes de executar o *Transporter*.

5.1.5 Classe CTimer

A classe CTimer é responsável pela execução de um temporizador. O temporizador serve para obter os valores do tempo em que o *Transporter* está a executar a captura, ou para retirar os valores de *timestamp*. Para a implementação do temporizador é necessário usar o relógio do computador. Para se poder utilizar o relógio do computador, estamos dependentes do sistema operativo onde a aplicação irá ser executada. Para implementar a classe CTimer é necessário fazer uma implementação do relógio para cada um dos sistemas operativos alvo, baseado nas API disponibilizadas por estes sistemas operativos.

A precisão do temporizador é importante para que se possa gerar eventos com intervalos de tempo correctos, caso se pretenda adicionar funcionalidades ao *Transporter* como a captura durante um intervalo específico de tempo. Quanto maior o número de bits do relógio do computador, maior a precisão; por este motivo a decisão foi utilizar temporizadores na ordem dos milissegundo ou micro-segundo.

O conjunto de bibliotecas Boost possui uma biblioteca que possibilita a utilização do temporizador, designada de Boost Timer [53]. No entanto, segundo a documentação da mesma, refere que a implementação desta função, embora tenha uma precisão alta, a latência na chamada era maior. Como pretendemos ter um temporizador que seja rápido e preciso, não é conveniente a utilização desta biblioteca, optando-se assim por usar API específicas de cada sistema operativo alvo.

Foram analisadas soluções para temporizadores de alta precisão para os sistemas operativos Windows, especificamente na versão Windows 7 32 bits e 64 bits, e no Fedora 14, a 64 bits. Para o sistema Mac OSX não foi possível analisar, dado que no presente momento não existe um computador Mac OS X disponível.

A classe CTimer está implementada, até a data, para suportar os sistemas operativos Windows e Linux. Fazendo uso da biblioteca Boost Signals2 [42], esta classe disponibiliza um sinal que é chamado sempre que decorre o tempo configurado. Com o Signal, qualquer classe que pretenda chamar algum método num determinado intervalo de tempo, pode registar esse método no sinal.

Quanto é feita a chamada ao método Start da classe CTimer, esta inicia uma *thread*, implementada com a biblioteca Boost Thread, que inicia a execução do temporizador. O método Stop permite parar a execução do temporizador.

A implementação do CTimer fazendo uso da API dos sistemas operativos, foi desenvolvida utilizando macros para diferenciar a nível de compilação o código a ser utilizado para cada um dos sistemas operativos. Esta decisão foi tomada dado que a implementação da classe ser pequena e assim retira a necessidade de criação de interfaces e implementação para cada um dos sistemas operativos alvo.

5.1.6 Timer no sistema operativo Windows

No sistema operativo Windows foram analisados vários métodos para obtenção de um temporizador preciso, nas quais as escolhidas para análise pela sua precisão e simplicidade de uso foram; *timeGetTime* [54], *GetTickCount* [55], *QueryPerformanceCounter* [56]. Apresenta-se na Tabela 12 uma comparação entre as 3 funções para decidir qual delas seria mais adequada.

Função	Características
TimeGetTime	<ul style="list-style-type: none"> - Precisão de 5 milissegundos - Retorna o tempo decorrido desde que o Windows iniciou. - Pode correr aproximadamente até 49,7 dias
GetTickCount	<ul style="list-style-type: none"> - Precisão de 10 a 16 milissegundos - Retorna o tempo decorrido desde que o computador iniciou. - Pode correr aproximadamente até 49,7 dias
QueryPerformanceCounter	<ul style="list-style-type: none"> - Em sistemas multi-processadores ou multi-core, não importa qual é o processador que foi chamado, muito embora possam ter valores de relógio diferentes. - Lê o valor do contador directamente do relógio do CPU, para ter maior precisão.

Tabela 12: Comparação das características das funções de temporizador disponibilizadas pela API do Windows

Dado que existe melhor precisão da função *QueryPerformanceCounter*, e o facto de ser independente do processador, torna esta função mais favorável para os requisitos da aplicação.

5.1.7 Timer no sistema operativo Linux

No caso do Linux, está disponível a função *clock_gettime* [57], que recebe como parâmetro a fonte de onde se pretende retirar o valor do tempo. Esta função tem uma resolução de segundos e nano-segundos. Para o corrente problema, dado que pretendemos o valor do tempo do sistema, usamos como fonte *clock_id* a opção *CLOCK_REALTIME* [57]. Esta é uma função em conformidade com a norma POSIX.1-2001.

5.2 Camada de Áudio

A camada de áudio realiza a ligação entre a placa de som e a aplicação. Esta camada é constituída pelas classes apresentadas na Tabela 13.

Classes	Descrição	Ficheiro
CAudioServer	Usa a biblioteca RTAudio para abstrair dos drivers e comunicar com a placa de som	AudioServer.h AudioServer.cpp
CAudioPort	Representa uma porta de captura ou <i>playback</i> no servidor	AudioPort.h AudioPort.cpp
CBuffer	Pertence à porta e é onde os dados são escritos e lidos	Buffer.h Buffer.Cpp
CBufferInfo	Descreve a informação relativamente ao <i>buffer</i>	Buffer.h

Tabela 13: Classes pertencentes à camada de áudio

O diagrama de UML da camada de áudio é apresentado na Figura 49.

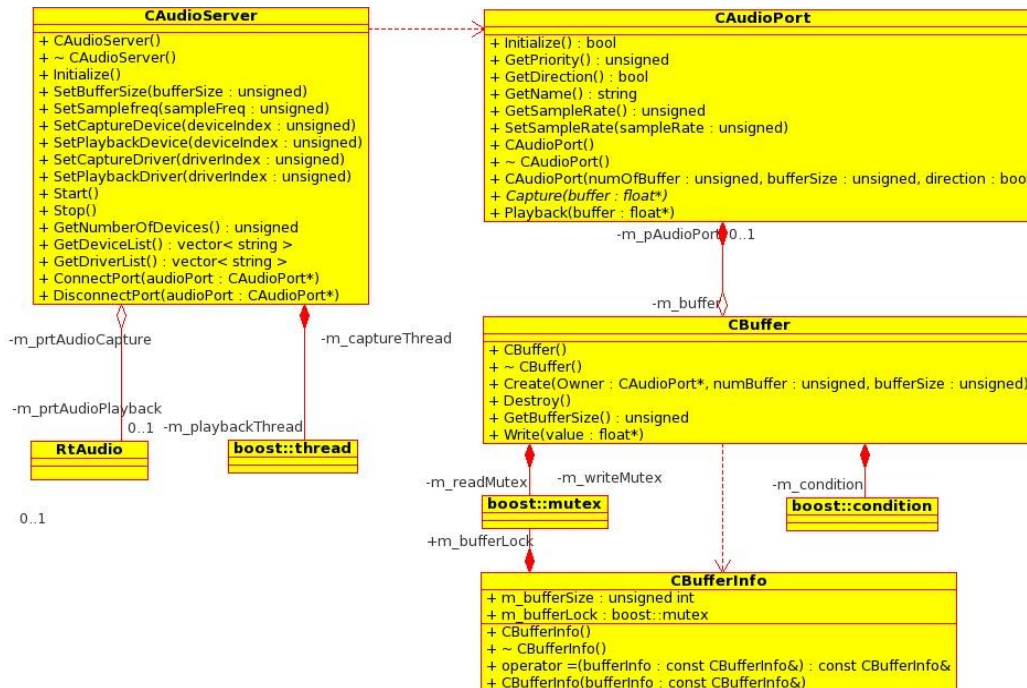


Figura 49: Diagrama UML da camada de captura áudio

A Figura 49 apresenta a estrutura de classes UML que fazem parte da camada de captura de áudio. Podemos ver associadas à estrutura de classes, mas sem muitos detalhes as classes RTAudio, boost::thread, boost::mutex e boost::condition, que são as classes de bibliotecas utilizadas para o desenvolvimento.

5.2.1 Classe CAudioServer

A classe CAudioServer é o componente para interacção entre a placa de som e a aplicação. É desenvolvido com a utilização da biblioteca RTAudio [30], as preocupações relativamente ao desenvolvimento dos *drivers* para diferentes sistemas operativos foram postas de parte. O RTAudio suporta *drivers* nos 2 sistemas operativos sendo este um dos requisitos do projecto.

A biblioteca RTAudio [30] também suporta canais áudio múltiplos, mas para a nossa aplicação só interessa um único canal. Dado que o projecto está ser desenvolvido em Fedora

14, e muito embora a biblioteca RTAudio suporte alguns *drivers* como *Advance Linux Sound Architecture* (ALSA) [34] e *Jack Audio Connection Kit* (JACK) [35] para áudio no Linux, na biblioteca será adicionada uma nova API que é o PulseAudio [36].

A arquitectura áudio no Linux, disponibilizada pelo ALSA não permite que mais do que uma aplicação tenha acesso a placa de som. O JACK veio resolver parte do problema, porque permite que várias aplicações tenham acesso à placa de som, e até mesmo redireccionar entradas e saídas de uma aplicação para outra. No entanto, o JACK é vocacionado mais para aplicações profissionais, que necessitam de baixa latência e tempo real, para além de que o JACK não é distribuído como fazendo parte da plataforma base do Linux, e o facto de que quando se utiliza o JACK nenhuma outra aplicação que não o suporte, tem acesso a placa de som.

O PulseAudio [36] veio resolver este problema: funciona como um *proxy* entre a placa de som e as aplicações. Faz uso do ALSA para ter acesso à placa de som, suporta baixa latência e tempo real. O PulseAudio também permite às aplicações usarem a placa de som mesmo com o JACK ligado, porque o suporta. O PulseAudio é já standard em várias distribuições Linux, o que motiva a inclusão desta API na biblioteca RTAudio.

O servidor áudio possui uma lista de portas com métodos para escrita e leitura. Estas portas são registadas no servidor, que as adiciona na lista de portas de captura ou portas de *playback*, dependendo da direcção escolhida se é de *input* ou *output*.

As portas possuem 1 ou mais *buffers*, para escrita ou leitura, permitindo a adaptação das portas caso se chegue à conclusão que o número de *buffers* para determinada porta não é suficiente. Esta opção é particularmente importante para os casos onde é necessário fazer processamento áudio, e há necessidade de não perder amostras recolhidas da placa de som.

5.2.2 Classe CAudioPort

CAudioPort é uma classe abstracta que representa uma porta de captura ou de *playback* que é conectada ao servidor áudio. Todos os componentes que pretendam receber sinal áudio ou enviar sinal áudio necessitam de registar a porta. Cada porta é constituída por *buffer* de escrita e de leitura. Os buffers estão implementados na classe CBuffer, que representa um conjunto de *n buffers*.

Os componentes que pretendam receber sinais do servidor áudio, derivam desta classe e implementam os métodos de escrita e leitura, Write e Read respectivamente. O método Read permite que o componente leia os sinais vindos do servidor áudio, enquanto que o método Write permite escrever no servidor áudio.

As portas têm prioridades, para definir a ordem onde será inserida a porta quando adicionadas à lista de portas existentes na classe CAudioServer. Cada porta possui uma única direcção, escrita ou leitura. Esta decisão é tomada pelo facto de podermos fazer captura de uma determinada placa de som e fazer a reprodução noutra placa de som diferente, caso o computador em questão tenha mais do que uma placa de som, algo que é muito comum em ambientes profissionais.

5.2.3 Classe CBuffer

Cada porta conectada ao servidor áudio é composta por um *buffer* da classe CBuffer. Como se pretende processar todo sinal áudio recebido, sem que se tenha perda de amostras, precisamos de garantir sincronismo entre a escrita do sinal e o processamento do mesmo. Dado que a escrita e leitura são executadas por diferentes *threads*, estamos na presença de um problema de sincronismo escritor/leitor.

A classe CBuffer suporta M buffers de tamanho N , criados mediante a passagem de parâmetros no método Cbuffer::Create. Esta opção permite que se possa implementar solução de *double buffering* ou *M buffering*. A solução de M buffers permite que se possa ter uma *thread* a escrever no *buffer* enquanto outra *thread* lê os dados do *buffer*.

A *thread* de leitura fica bloqueada à espera que seja notificada pela *thread* de escrita assim que a escrita num determinado *buffer* termina. Em seguida, a *thread* de escrita passa para o *buffer* seguinte enquanto a *thread* de leitura escreve no *buffer* anterior. A escrita e leitura nos *buffers* é feita sempre de forma sequencial, assim, logo que uma *thread* termina de processar um *buffer*, incrementa o número do *buffer* para aceder ao próximo. A Figura 50 ilustra a primeira iteração entre o leitor e escritor.

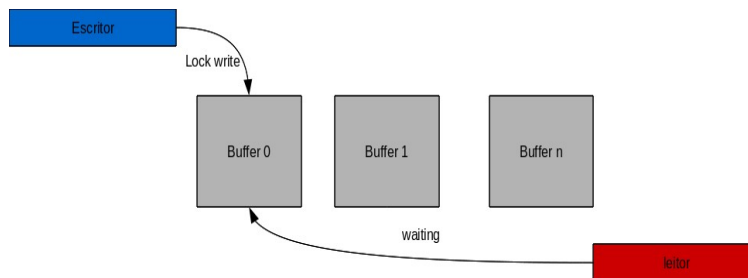


Figura 50: 1ª iteração entre o leitor e escritor

Na segunda iteração o escritor notifica o leitor de que já pode ler o *buffer* 0, libertando o *lock* do *buffer*, e em seguida passa para o *buffer* seguinte, ilustrado na Figura 51.

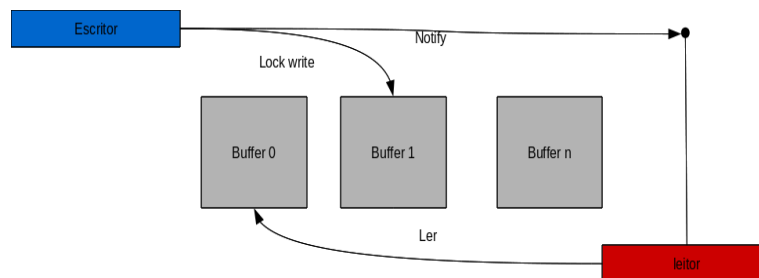


Figura 51: 2ª iteração entre leitor e escritor

O processo repete-se até chegar ao último *buffer*. Dependendo da velocidade de leitura do leitor, este poderá estar alguns *buffers* atrás do escritor. Quando chega ao último *buffer*, a *thread* escritora volta para o início do *buffer* começando o processo novamente.

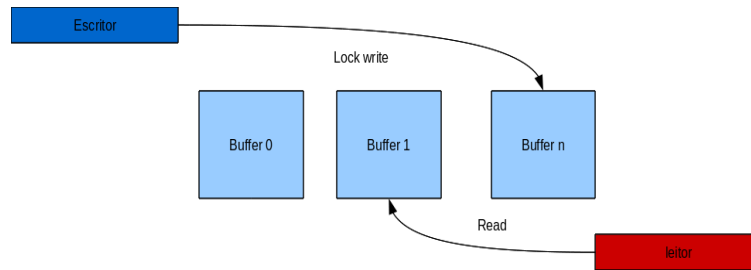


Figura 52: M iteração entre leitor e escritor

Dado que se pretende ter um número de *buffers* configurável, é necessário que cada *buffer* tenha associado um *lock*. Está definido que a escrita e leitura nos *buffers* é feita de forma sequencial, o que significa que logo após a escrita ou leitura num *buffer*, a *thread* move-se para escrever ou ler o *buffer* seguinte. Quando qualquer das *thread* tenta aceder ao *buffer*, verifica se este *buffer* está cheio ou vazio, para escrita e leitura respectivamente e fica em espera. Também pode ocorrer que a *thread* que pretenda aceder ao *buffer* encontre o *buffer* bloqueado porque outra *thread* já está a aceder; tal é um comportamento expectável e necessário para evitar conflitos de escrita e leitura nos dados.

Toda a implementação relacionada com os objectos de sincronismo foi desenvolvida utilizando a biblioteca Boost Thread [40].

A principal vantagem de ter o número de *buffer* configurável está na possibilidade de adaptar cada porta à necessidade na qual está conectada ao servidor áudio. Um exemplo, é a necessidade de escrita no disco do sinal captado pelo servidor áudio. Dado que a escrita no disco poder ser mais lenta que, por exemplo, o processamento do sinal, existe maior necessidade de aumentar o número de *buffers* e assim dará tempo suficiente para que a operação de escrita no disco ocorra sem que se tenha perda de amostras do sinal.

5.3 Camada de processamento de sinal

A camada de processamento áudio é responsável por todo o processamento de sinal efectuado na aplicação. Implementa todos os métodos necessários para a aplicação. Esta camada é composta na realidade por um conjunto de classes e funções para estimação de frequência, transformação de notas musicais e cálculo da duração de notas. A camada de processamento áudio é composta pelas classes apresentadas na Tabela 14.

Classes	Descrição	Ficheiro
CSignalProcessing	Usa a biblioteca FFTW para cálculo da DFT e possui métodos para processamento de sinal, FFT, Hamming, MaximumPeak	SignalProcessing.h SignalProcessing.cpp
XmlReport	Escrita do ficheiro de <i>benchmark</i> em xml	Benchmark.h

Tabela 14: Classes pertencentes à camada de processamento áudio

O diagrama UML da camada de processamento áudio é apresentado na Figura 53.

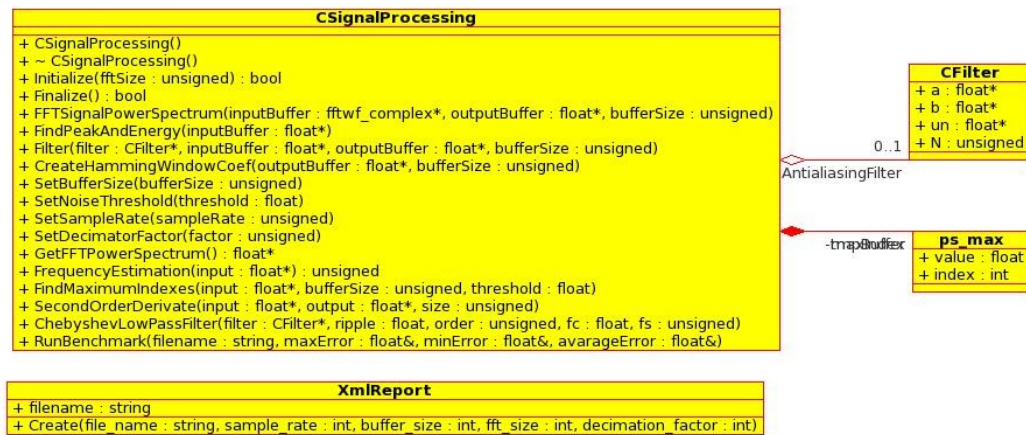


Figura 53: Diagrama das classes da camada de processamento de sinal

Na Figura 53 observa-se no diagrama UML da classe `CSignalProcessing` todos os métodos disponíveis para processamento de sinal. A classe inclui filtros passa-baixo, estimação de frequência usando FFT, cálculo da energia do sinal, janela de Hamming, decimação e cálculo de pico máximo e mínimo de energia.

5.3.1 Classe `CSignalProcessing`

A classe `CSignalProcessing` implementa os vários métodos e algoritmos para fazer processamento de sinal áudio. O conjunto de soluções para processamento de sinal, contempla estimação de frequência com algoritmo FFT usando a biblioteca FFTW, aplicação de janela de Hamming, cálculo da potência do espectro do sinal e duração das notas.

A biblioteca FFTW calcula a FFT a N pontos. A biblioteca utiliza vários algoritmos, o Cooley-Tukey, o Radix para números primos e Split-Radix [58]. A FFTW calcula a FFT sobre sinais representados na seguinte forma tempo-frequência: com números complexos-complexos, real-complexo e real-real. Para esta solução foi utilizada o cálculo da FFT real-complexo. O resultado da FFTW é um *array* com simetria hermitiana, ou seja,

$$Y[k] = \bar{Y}[k + N/2] \quad k = 0, 1, \dots, N, \quad (27)$$

onde N é o número de pontos, porque o sinal é real no domínio do tempo.

Para estimar a frequência detectada calculamos o módulo do espectro pela FFT, e procuramos o índice onde este contém o valor máximo. Dada a simetria do *array* de potência do espectro, só é necessário pesquisar pelos máximos em metade do *array*. Dado que podemos detectar várias frequências, a função que pesquisa pelos máximos, acumula os valores máximos encontrados cuja potência do sinal seja maior que um determinado limiar passado como parâmetro. No máximo, podem ser tocadas 6 notas numa guitarra, formando um acorde, o que significa que podemos detectar no máximo 6 frequências em simultâneo.

Outra características na estimação da frequência está relacionada com a definição de uma frequência máxima e mínima. Esta definição permite otimizar a pesquisa no *array* de potência do espectro, por intermédio dos índices correspondentes a frequência máxima e mínima. Desta forma a pesquisa inicia-se no índice máximo.

Para fazer *down-sampling* do sinal usando decimação, a classe de processamento de sinal possui um filtro passa-baixo, implementação do filtro de Chebyshev tipo I estudado no capítulo 3 secção 3.4.1. O filtro é aplicado depois da recepção do sinal áudio para remover as frequências que não cumpram com a regra de Nyquist antes de aplicar a decimação. A frequência de corte do filtro passa-baixo é parametrizável; para o caso da guitarra eléctrica, os valores são entre 82 Hz a 1175 Hz, o que significa que o filtro não deve cortar estas frequências.

A classe está desenvolvida por forma que se possa fazer a chamada das várias funcionalidades de forma separada.

5.4 Camada de gravação

É importante que o guitarrista possa ouvir toda a música que tocou, e por este motivo a aplicação tem uma camada responsável por gravar todo o sinal vindo da guitarra eléctrica num formato áudio.

Esta camada não foi incluída na arquitectura porque ela por si só representa uma funcionalidade nova e não faz necessariamente parte da solução a ser implementada. No entanto o facto de ser possível gravar o sinal num formato áudio, torna possível também a utilização do mesmo sinal para poder fazer um processamento posterior de todo o sinal. Também abre possibilidades para os casos nos quais foram aplicadas algumas técnicas de guitarra como *bending*, se possa futuramente analisar e detectar estas técnicas ou mesmo o guitarrista corrigir e adicionar estas técnicas na tablatura. A Tabela 15 apresenta os ficheiros que implementam a camada de gravação.

Classe	Descrição	Ficheiro
CAudioFileIO	Usa a biblioteca libsndfile para gravar o sinal de entrada num ficheiro no formato áudio.	CAudioFileIO.h CAudioFileIO.cpp

Tabela 15: Classes e ficheiros pertencentes à camada de gravação áudio

5.4.1 Classe CAudioFileIO

A classe CAudioFileIO é responsável por gravar o sinal áudio num ficheiro. Esta classe está implementada com a biblioteca libsndfile [48]. Esta biblioteca suporta a gravação de ficheiros áudio em vários formatos (WAV, FLAC, OGG entre outros).

O ficheiro áudio criado é protegido por um *lock* de escrita exclusivo, para evitar que outro processo ou *thread* possa escrever no ficheiro, mas permite ter outros processos ou *threads* a ler do ficheiro. O diagrama UML correspondente a esta camada é ilustrado na Figura 54.

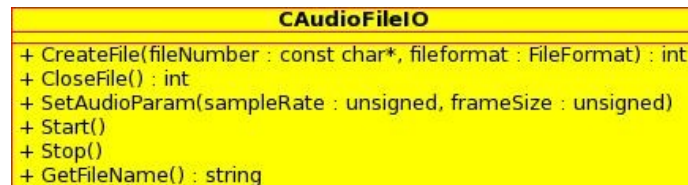


Figura 54: Diagrama UML da classe CAudioFileIO

Na Figura 54, apresenta-se o diagrama UML da classe CAudioFileIO. Podemos observar os métodos Start e Stop que iniciam e param o processo de captura respectivamente. O método Start cria uma thread para fazer a captura. Como a classe CAudioFileIO deriva também de AudioPort, o servidor áudio fará a chamada do método virtual Capture passando o *buffer* com amostras do sinal como parâmetro. Na implementação do método virtual Capture a classe CAudioFileIO chama o método Save que grava o sinal em formato áudio, usando a biblioteca libsndfile.

A biblioteca libsndfile para gravar ficheiros precisa do parâmetro tamanho total do ficheiro. Por este motivo foi necessário utilizar uma estratégia para gravar continuamente o som capturado da guitarra eléctrica. A estratégia foi gravar cada *frame* no final do ficheiro. A biblioteca libsndfile possui esta funcionalidade, bastando por cada *frame* que se recebe uma amostra gravar o ficheiro. Desta forma é possível a escrita no ficheiro em tempo real.

A gravação no ficheiro também possibilita um processamento posteriormente ao guitarrista ter finalizado de tocar, o que permite corrigir possíveis erros na detecção. Quando o guitarrista pára de tocar, torna possível o processo de detecção de frequências correr em modo *offline*, e se possível com parâmetros que permitam melhorar a precisão da detecção das notas musicais.

5.5 Integração entre servidor e interface gráfica

A aplicação tem duas componentes principais: o servidor, que faz a captura e transformação dos sinais em notas musicais; a interface gráfica que permite ao utilizador visualizar as notas que estão a ser tocadas. Cada componente executa-se em processos diferentes, o que torna necessário desenvolver uma estratégia para que possa haver interacção entre o servidor áudio e o ambiente gráfico. O servidor áudio, é uma aplicação *standlone* que é executada pela interface gráfica, que vai efectuar todos os processos de captação do sinal áudio e conversão em notas musicais. A interface gráfica permite ao utilizador iniciar ou parar o processo de captura do sinal; neste sentido existem duas formas de comunicação entre a interface gráfica e o *GSTTransporter* que são as seguintes: ficheiro áudio e fila de mensagem. A Figura 55 ilustra o diagrama de integração entre o servidor e a interface gráfica.

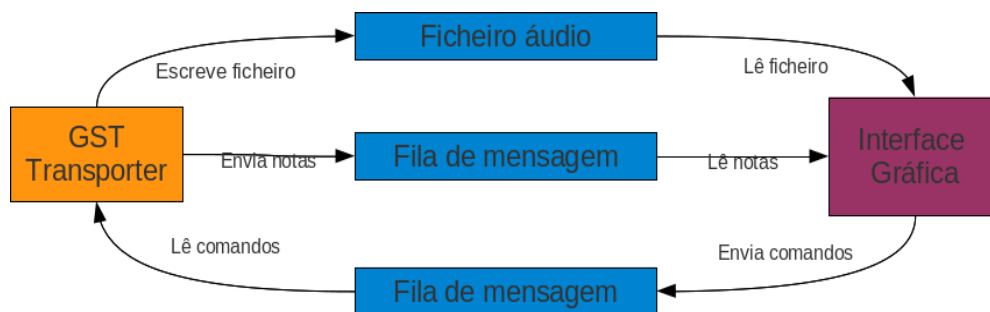


Figura 55: Diagrama de integração entre o GSTTransporter e a Interface gráfica.

O servidor áudio é iniciado por meio de passagem de parâmetros ao programa executável. Quando o servidor áudio inicia a execução, todo o sinal áudio é gravado num ficheiro no formato especificado pelo utilizador por meio de parâmetro. Os comandos que permitem iniciar a captura são enviados por meio de fila de mensagem. As notas detectadas são armazenadas numa fila de mensagens partilhada entre o servidor áudio e a interface gráfica.

Resumindo, a solução para integrar o servidor e a interface gráfica passa pela utilização de métodos de comunicação entre processos. Para implementação destes métodos é utilizada a biblioteca Boost Interprocess [51]. Esta biblioteca implementa vários métodos de comunicação entre processos.

Para além disso, é necessário integrar o código do *Transporter* desenvolvido em C/C++ com o da interface desenvolvido em C#. Para este efeito foi necessário desenvolver uma camada de integração entre C/C++ e o C#.

A necessidade de integração de código C++ tinha que ser resolvida de forma que se tivesse o menor número de pontos de integração possível, mantendo a simplicidade da sua implementação. O problema de integração levantou-se quanto foi necessário aceder à memória partilhada e à fila de mensagem criada usando a biblioteca Boost interprocess, porque não há mecanismos para aceder a esta memória directamente da plataforma .NET.

A comunicação entre processos na plataforma .NET pode ser feita utilizando Named Pipes, Remoting, ou mesmo MicroSoft Message Queue (MSMQ). Mas para o código C++ aceder a estes mecanismos de comunicação entre processos envolveria uma maior complexidade e custo de desenvolvimento. A opção recaiu em desenvolver uma biblioteca dinâmica em C que iria ser chamada pelo código C#.

Dado que a comunicação entre o *Transporter* e a interface gráfica é feita através de fila de mensagem. O número de funções a serem exportadas para C# não são muitas, e o ponto de integração é único. Neste sentido, de forma resumida é necessário criar uma forma de o código desenvolvido em C# possa aceder à fila de mensagem desenvolvida em C++. Para tal ser possível, foi criada uma biblioteca dinâmica em C/C++, que implementa o acesso a fila de mensagem desenvolvida em C++ com a biblioteca Boost interprocess, para ler e escrever na fila de mensagem. Recorre-se à plataforma .NET, utilizando a InteropServices do *namespace* System.Runtime.InteropServices [59]. O InteropServices permite carregar bibliotecas desenvolvidos em código nativo. O atributo DllImport [60] permite carregar uma biblioteca dinâmica implementada noutra linguagem e criar um *entry point* para o método implementado na DLL.

Como a aplicação está implementada para suportar vários sistemas operativos, a biblioteca dinâmica tem uma extensão diferente para cada sistema operativo. No caso do Linux as bibliotecas possuem extensão *.so* e na plataforma Windows *.dll*. Por este motivo na biblioteca criada para integração não terá uma extensão nem *.so* ou *.dll*. Desta forma não será necessário criar um nome separado para cada plataforma. A diferença no formato binário gerado em cada sistema operativo implica que a biblioteca dinâmica terá que ser compilada separadamente em cada sistema operativo.

A biblioteca criada chama-se *Transporter* e implementa métodos que permitem interacção entre o *GSTTransporter* e a interface gráfica. Os métodos com implementação em C++ têm que ser expostos como sendo *extern "C"* porque assim podem ser chamados como funções C, o que facilita a importação. Quando o método é compilado como código C++, o nome é alterado e a estrutura da chamada o método pode também ser alterada dependendo do compilador. Depois da biblioteca criada em C/C++, uma classe em C# foi criada que importa todos os métodos para .NET, permitindo assim o código C# chamar código nativo C/C++.

No entanto, testes desta solução revelaram que muito embora a biblioteca dinâmica estivesse a exportar as funcionalidades do código C++ como código C, as chamadas nos métodos internos da biblioteca *Boost message_queue* fazem chamada a estruturas complexas em C++, o que não permitia que esta solução fosse utilizada, e por este motivo foi necessário implementar outra solução.

A solução utilizada passou pela criação de um programa em C++ que é responsável por ler e escrever na fila de mensagens. Este programa é um executável que recebe como parâmetros o número da mensagem a ser enviada e faz a escrita deste valor na fila de mensagem de comandos. Quando esta aplicação é executada sem parâmetro, faz a leitura da fila de mensagem que contém as *frames* e escreve no *standard output* no formato XML. Desta forma a aplicação que executa tem que redireccionar a saída do *standard output* para ler e receber a *frame* no formato XML. É possível observar a estrutura UML de uma *frame* na secção 5.1.3 deste Capítulo, Figura 48. O objecto do tipo *Frame*, é escrito como um objecto em XML no *standard output* mediante a implementação do operador de escrita "<<". Desta forma o programa responsável pela leitura da *frame* na fila de mensagem e posterior escrita no *standard output* escreve o objecto como uma estrutura XML.

5.5.1 Integração com ficheiro áudio

O ficheiro áudio é criado e escrito pelo *GSTTransporter* enquanto faz a captação do sinal vindo da guitarra eléctrica, permitindo enquanto isso a interface gráfica fazer a leitura do ficheiro para apresentação na aplicação. O ficheiro é bloqueado para escrita não permitindo que outro processo possa escrever no ficheiro enquanto o *GSTTransporter* executa a escrita. Para implementar o *lock* no ficheiro é utilizada o *Boost File Lock* [61]. O *lock* é importante também para não permitir que qualquer outro processo por exemplo tente apagar o ficheiro enquanto este está a ser escrito.

Dado que a aplicação guarda um ficheiro contendo todo o sinal tocado pelo guitarrista, é possível, por exemplo, fazer uma detecção mais afinada das notas tocadas pelo guitarrista no modo *offline*. Este processamento pode ser mais demorado e utilizar parâmetros na

detecção de frequência mais precisa. Assim, o resultado deste último processamento pode ser comparado com o do processamento anterior, permitindo uma comparação e correcção por parte do utilizador.

5.5.2 Integração com fila de mensagem

A fila de mensagem é utilizada para passagem de comandos entre a interface gráfica e o *Transporter* e também para o envio de *frames* contendo informação das frequências detectadas. A fila de mensagem é implementada usando Boost *message_queue* [38]. A classe *message_queue* possui 3 métodos para envio e 3 métodos para recepção de mensagens, *send*, *try_send*, *timed_send*, *receive*, *try_receive* e *timed_receive*. Para o desenvolvimento da solução são utilizados os métodos *send* e *receive*, os quais ficam bloqueados se a fila de mensagem estiver cheia ou vazia, respectivamente. Os métodos que iniciam com *try_* e *timed_*, são métodos que não bloqueiam quando a fila está cheia ou vazia no *send* e *receive*.

Para a aplicação é necessário que os métodos fiquem bloqueados enquanto não existem mensagens na fila no caso da *thread* que faz a recepção dos comandos, e no caso de envio de *frames* o método utilizado será não bloqueante. No *Transporter* existe uma *thread* para recepção dos comandos vindos da interface gráfica que mediante o código recebido executa determinadas acções.

Tal como referido no início deste Capítulo, é a interface gráfica que faz o envio de comandos ao *GSTTransporter*, e este último é responsável pela execução deste comando. No início deste Capítulo foi apresentada integração entre a interface gráfica, que está desenvolvida em C# com a fila de mensagens desenvolvida em C/C++. As mensagens enviadas pela interface gráfica são valores do tipo inteiro que são processadas na aplicação.

O problema relativo ao acesso às filas de mensagem foi resolvido usando um processo que faz a execução de outro processo, onde o último, desenvolvido em C++, é responsável por ler e escrever na fila de mensagem. Para a interface gráfica enviar mensagens e receber as notas foi desenvolvido o projecto *xcomm*, que consiste numa aplicação que quando executado com parâmetros, escreve este valor na fila de mensagem correspondente aos comandos. Quando executado sem parâmetros o *xcomm* lê o conteúdo da fila de mensagem que contém as *frames* do *Transporter* e escreve no *standard output*, em formato XML.

No entanto, esta solução não é satisfatória a nível de implementação, requerendo a análise de outra solução. Uma investigação na biblioteca Boost *message_queue*, mostra que a fila de mensagem é implementada utilizando mapeamento de memória em ficheiro. Num trabalho futuro o objectivo será desenvolver uma estrutura em C# com as mesmas características da Boost *message_queue* e assim permitir acesso directo entre a fila de mensagem Boost e o Mono/.NET. A implementação em Mono/.NET pode ser realizada com a classe *MemoryMappedFile* [62]. Esta classe permite mapear o conteúdo de um ficheiro no espaço de endereços de uma aplicação, realizando a escrita e leitura de dados no ficheiro.

5.6 Interface gráfica

Nesta secção analisa-se o desenvolvimento da interface gráfica. A interface gráfica é a forma do guitarrista interagir com a aplicação, e foi desenvolvida utilizando Mono, C# e GTK-Sharp [63]. O objectivo da interface gráfica é permitir a visualização das notas detectadas em forma de tablatura, ter uma percepção visual do estado da aplicação e interagir com o *Transporter*. Alguns componentes gráficos, tais como a pista áudio e tablatura, tiveram que ser criados para permitir apresentar informação mais no âmbito de uma aplicação multimédia.

Pretendia-se que a tablatura inicial fosse em modo texto, mas existia maior complexidade na apresentação em formato texto do que num formato gráfico. Foram pesquisadas bibliotecas que permitissem em tempo real a representação de notas musicais, tanto em forma de tablatura como de partitura, na aplicação gráfica. Como resultado da pesquisa foram encontradas duas soluções *Open Source* para criação de notação musical com base num ficheiro texto, que contém instruções cuja compilação resulta numa partitura. Algumas das soluções encontradas foram LilyPond [64], MusicXML [65], ABC Notation [66] e VexFlow [67]. Estas API baseiam-se na criação de partituras usando um ficheiro de texto escrito com uma gramática específica e depois são compilados para sua apresentação. No projecto pretende-se que as notas sejam mostradas logo que sejam desenhadas mas com formato mais simples, visto que seria necessário um ciclo de aprendizagem para adaptar a estruturas das várias API apresentadas.

A solução para a tablatura foi de criar um componente gráfico que desenha a tablatura conforme recebe as notas. Os componentes pertencentes à interface gráfica são apresentados e descritos na Tabela 16.

Classes	Descrição	Ficheiro
Main	Classe principal que inicia a aplicação	Main.cs
Metronome	Janela que apresenta o metrónomo	Metronome.cs
GSTMainWindow	Janela principal	GSTMainWindow.cs
GuitarTuner	Janela que apresenta o afinador de guitarra	GuitarTuner.cs
SplashScreen	Janela inicial da aplicação enquanto carrega	SplashScreen.cs
SessionDocument	<i>Widget</i> que representa um sessão composta por editor de tablatura e pista de áudio	SessionDocument.cs
DlgSessionName	Caixa de dialogo para introdução do nome e parâmetros da sessão	DlgSessionName.cs
LibSndFile	Classe para leitura e escrita de ficheiros áudio	LibSndFile.cs
TabScoreEditor	<i>Widget</i> para desenho da tablatura	TabScoreEditor.cs
TimeLineBar	<i>Widget</i> para desenho da barra de tempo	TimeLineBar.cs
TabPageHeader	<i>Widget</i> para <i>header</i> da tabulação de sessões	TabPageHeader.cs
Transporter	Classe que implementa funcionalidade de comunicação com o <i>Transporter</i>	Transporter.cs
TransporterControl	<i>Widget</i> para controlar o <i>Transporter</i> , iniciar e parar a captação	TransporterControl.cs

Classes	Descrição	Ficheiro
Fretboard	<i>Widget</i> que representa o braço de uma guitarra	Fretboard.cs
AudioTrack	<i>Widget</i> que implementa uma janela para mostrar em formato WAVE o sinal áudio	AudioTrack.cs
AudioFormat.cs	Definição do formato áudio	AudioFormat.cs
DataChunk	Classe que representa os <i>bytes</i> do ficheiro áudio	DataChunk.cs
FMTChunk	Classe da especificação do ficheiro tipo WAVE	FMTChunk.cs
RiffChunk	Classe da especificação do ficheiro tipo WAVE	RiffChunk.cs
WaveFile	Classe que representa um ficheiro WAVE	WaveFile.cs
WaveFileReader	Classe para leitura de um ficheiro WAVE	WaveFileReader.cs
WaveFileWriter	Classes para escrita de ficheiro WAVE	WaveFileReader.cs
TabScore	Representa a estrutura de uma tablatura	TabScore.cs
Frame	Classe que representa uma <i>frame</i> do <i>Transporter</i>	Transporter.cs
SessionDocument	<i>Widget</i> que representa um sessão composta por editor de tablatura e pista de áudio	SessionDocument.cs
DlgSessionName	Caixa de dialogo para introdução do nome e parâmetros da sessão	DlgSessionName.cs
StaffNote	Classe que representa uma conjunto de notas da tablatura	StaffNote.cs
Freq2NoteUtil	Classe estática para conversão de frequências em notas musicais	Freq2NoteUtil.cs
AudioLevelMeter	<i>Widget</i> para desenhar o nível do sinal áudio	AudioLevelMeter.cs
AudioKnob	<i>Widget</i> para botões de controlo de volume	AudioKnob.cs
AudioSlider	<i>Widget</i> para desenho de <i>slider</i> para controlo de volume	AudioSlider

Tabela 16: Classes pertencentes à interface gráfica

Para uma visão global da aplicação, a Figura 56 apresenta a estrutura da aplicação organizada em blocos, onde se observa a interacção que cada camada tem e as funcionalidades que estão inerentes a cada uma.

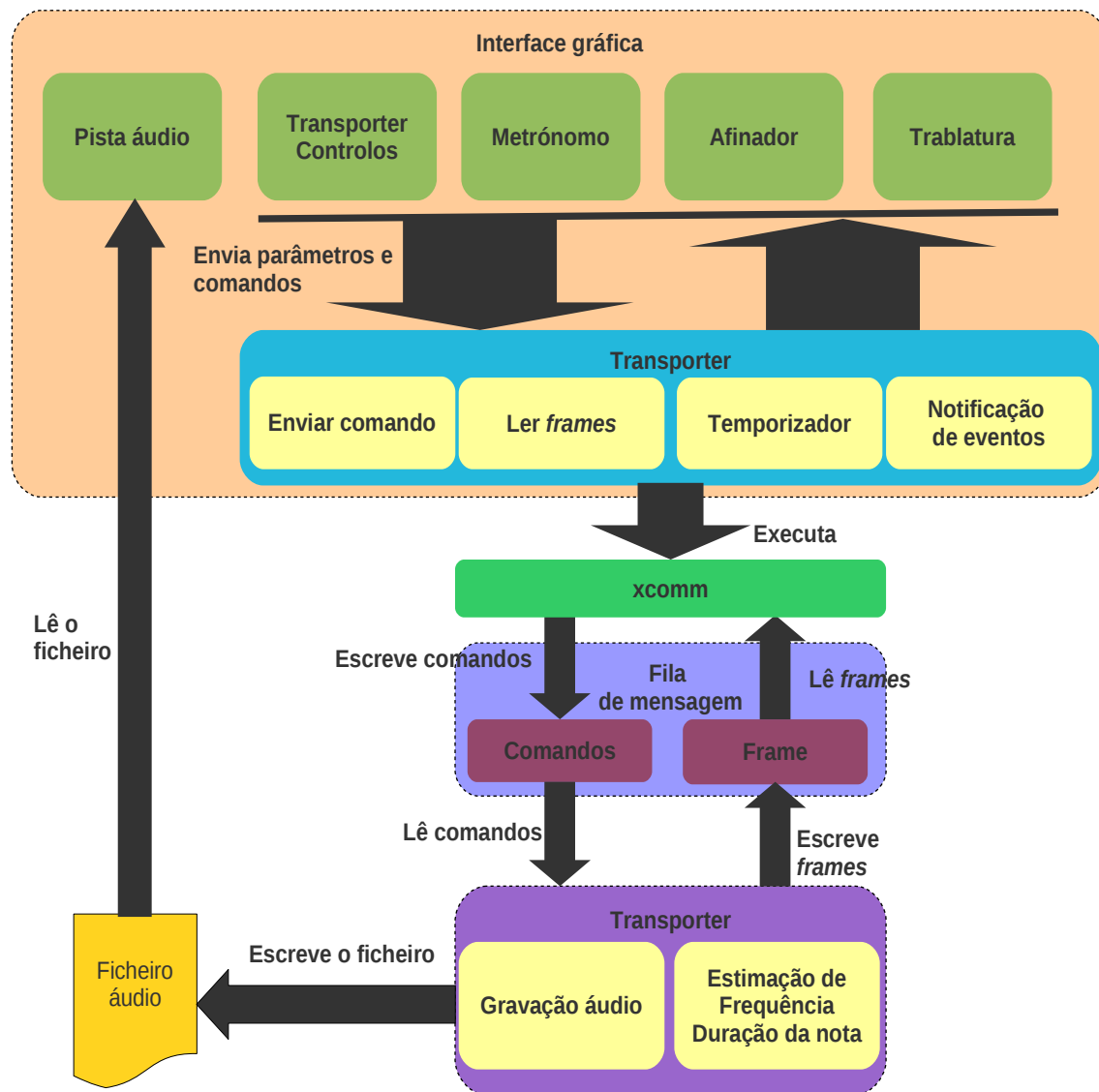


Figura 56: Diagrama geral da aplicação

O diagrama na Figura 56 pretende clarificar principalmente o modo como a interface gráfica está organizada e como é a interação entre os vários componentes gráficos. Em secções anteriores foi descrito o desenvolvimento do *Transporter*, agora é necessário apresentar o funcionamento da interface gráfica. Dentro do bloco interface gráfica os blocos verdes representam classes que recebem notificação de eventos da classe *Transporter*.

Observamos ainda que o componente *Pista de áudio* não tem ligação ao *Transporter*. A pista áudio faz uso do evento de actualização de um ficheiro para poder ler o ficheiro e desenhar a forma de onda do sinal áudio.

5.6.1 Classe Transporter

O *Transporter* é uma classe com métodos estáticos que implementa na interface gráfica as interações com o *Transporter*. Todas as *frames* recebidas pelos componentes gráficos são lidos pela classe *Transporter* que notificará por meio de evento as classes inscritas na recepção de *frames*. Esta classe possui um temporizador de intervalo configurável em

milissegundo que define o período de tempo que deve ler uma *frame* da fila de mensagem. O temporizador é accionado quando seleccionado o método Start, e no intervalo definido retira uma *frame* no formato XML da fila de mensagens serializa-a, transformando-a num objecto do tipo Frame. Em seguida notifica este evento a todas classes inscritas no evento passando como parâmetro o objecto Frame criado. A Figura 57 apresenta o diagrama UML da classe Transporter.



Figura 57: Diagrama UML da classe GST.Transporter

Esta classe possui um membro do tipo `ProcessStartInfo` [68]. Este membro permite passar a informação relativa a um processo para ser executado. Este membro é preenchido com informação do executável do xcomm, uma das propriedades do `ProcessStartInfo` é o redireccionamento do *standard output* de um processo. Para o nosso caso é necessário o redireccionamento do *standard output* porque precisamos receber a *frame* no formato XML da aplicação xcomm. Desta forma quando a aplicação xcomm escrever no *standard output* é possível receber os dados escritos.

O intervalo de tempo para actualização é definido à medida do modo que o guitarrista estiver a funcionar, modo afinador ou de transcrição musical. No caso de estar a utilizar o modo afinador o intervalo de actualização dos botões LED da interface gráfica (Figura 61) é feita com base na percepção visual que o humano tem de uma imagem em tempo real. Neste sentido realiza-se uma actualização num intervalo de 20 a 30 milissegundos. No caso que se está a proceder à captura das amostras o intervalo de tempo será baseado nos parâmetros do metrónomo, usando as equações (39), (40), (41) e (40) apresentadas no Capítulo 3 secção 3.6 para o cálculo do intervalo.

5.6.2 Classe libSndFile

O `libSndFile` é uma classe com métodos estáticos que implementa métodos para escrita de leitura de ficheiros áudio. Dado não existir uma biblioteca que permita a leitura e escrita de ficheiros áudio em .NET, que não dependa directamente do sistema operativo, a opção passou por utilizar os serviços de interoperabilidade do .NET e exportar as funcionalidades

existentes na biblioteca dinâmica libSndFile. A Figura 58 apresenta o diagrama UML da classe libSndFile.

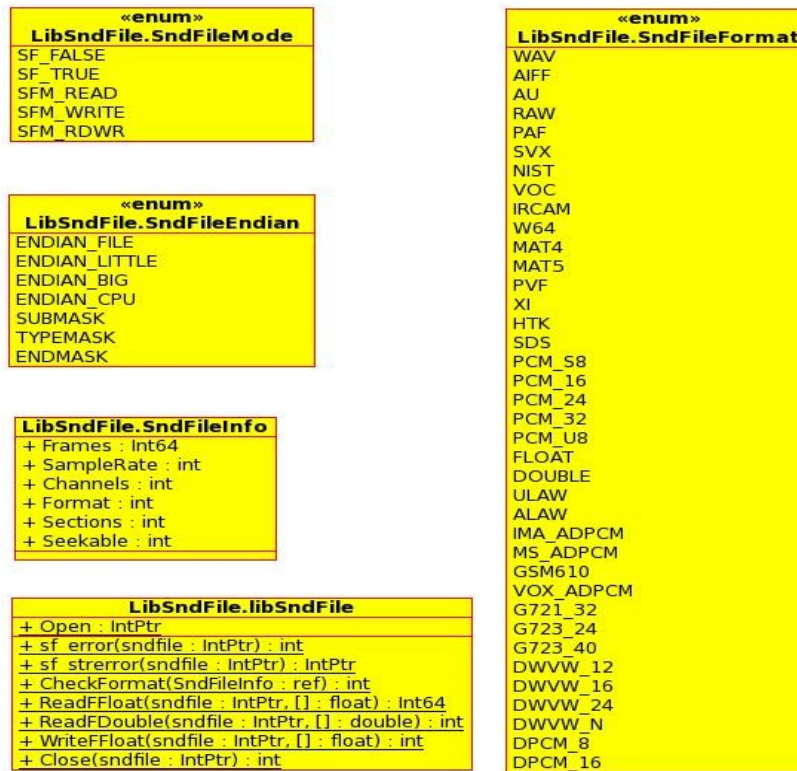


Figura 58: Diagrama UML do namespace LibSndFile com classes e enumerados

Podemos observar na Figura 58 que, nem todos os métodos existentes na biblioteca dinâmica libSndFile foram implementados, apenas os necessários para permitir a leitura de ficheiro. Esta classe será utilizada para fazer a leitura do ficheiro áudio pelo componente que desenha a pista áudio.

Para importação do código em C para C# foi utilizado o atributo `DllImport` [69], especificando o nome do *EntryPoint* na biblioteca dinâmica libSndFile. Para a estrutura SndFileInfo usou-se o atributo `StructLayout` [70] definido como sequencial para permitir o mapeamento desta estrutura com a estrutura existente na biblioteca dinâmica.

No método `Open`, é necessário retornar um apontador que corresponde ao apontador para o ficheiro no código *unmanager* este apontador corresponde ao tipo `IntPtr` [71] no código C#. Este método também recebe um parâmetro do tipo `string` que representa o nome do ficheiro a ser aberto, para ser reconhecido na biblioteca dinâmica, visto que essa espera um apontador de caracteres, é necessário usar `MarshalAs` [72], com `UnmanagedType.LPStr` [72], que converte a `string` em código *managed* para um apontador de caracteres em código *unmanaged*.

5.6.3 Janelas da aplicação

O ecrã principal da aplicação, apresentado na Figura 59, permite interação do guitarrista com a aplicação e a visualização das notas tocadas pelo guitarrista.

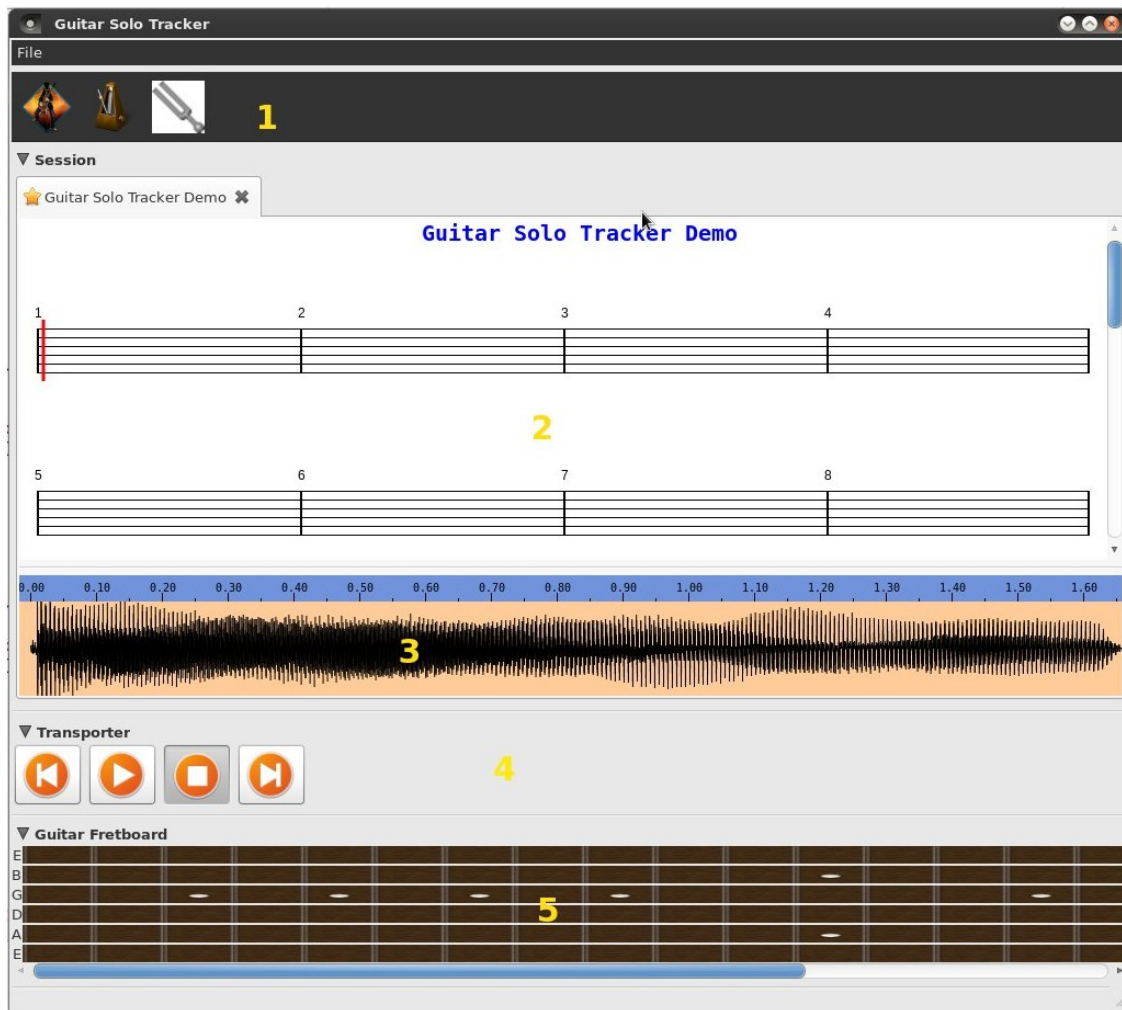


Figura 59: Guitar Solo Tracker janela principal

A interface gráfica tem uma barra de ferramentas (1 na Figura 59) onde disponibiliza botões para abrir uma nova sessão, metrónomo e o afinador de guitarra (2 na Figura 59), onde serão apresentadas as notas detectadas pela aplicação. Em 3 é a área de apresentação do sinal áudio em forma de onda. A apresentação deste sinal é feita com base no ficheiro que é gravado enquanto o guitarrista toca a guitarra. A área 4 contém os controlos do *Transporter* que permitem iniciar e parar todo o processo de captação do sinal. Na área 5 apresenta-se o braço da guitarra, que permite a alteração e edição das notas que estiverem erradas na detecção. A edição das notas só poderá ser feita quando a aplicação não estiver em modo de captura.

Quando o guitarrista inicia a aplicação, é criada uma sessão. Uma sessão no Guitar Solo Tracker representa o conjunto de notas detectadas de áudio gravado, que na Figura 59 estão indicados como 2 e 3 respectivamente. Neste sentido, o guitarrista pode criar várias sessões e por cada uma tocar a música e detectar as notas.

Quando é seleccionado o menu do metrónomo na barra de ferramentas, esta abre a janela do metrónomo. Esta janela permite definir os valores de BPM, o número de batidas, ligar e desligar o metrónomo. A Figura 60 mostra o metrónomo da aplicação.

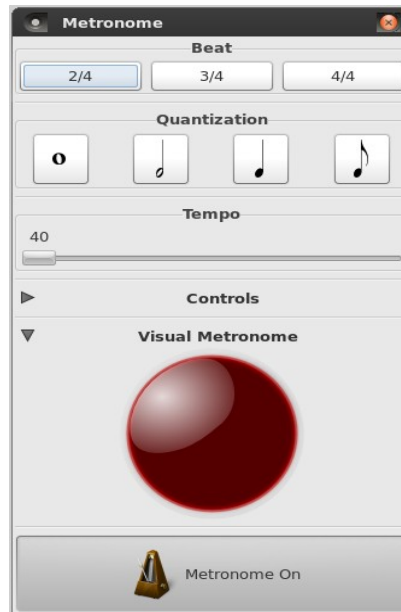


Figura 60: Janela do Metrónomo

Quando o guitarrista escolhe o valor do tempo, o tipo de batida e liga o metrónomo fazendo *click* no botão “Metronome On”, este permite visualizar os batimentos do metrónomo. Os parâmetros escolhidos no metrónomo são enviados para o *Transporter* antes de iniciar a captura. Estes parâmetros são importantes para a estimação da duração da nota. O tempo determina a velocidade e a batida a acentuação da nota quando o metrónomo está em execução. A quantização vai determinar o número de notas assumida em cada batimento de metrónomo. Os dados do metrónomo só podem ser alterados quando o *Transporter* não estiver em execução.

O menu Guitar Tuner quando seleccionado abre a janela do afinador de guitarra. A Figura 61 apresenta a janela do afinador de guitarra.

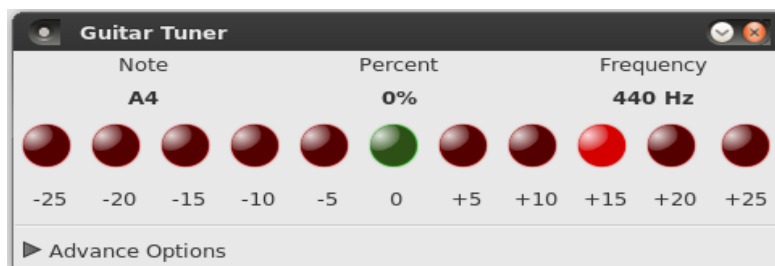


Figura 61: Janela do afinador de guitarra

O guitarrista quando pretende afinar a guitarra, basta começar a tocar as cordas e automaticamente os botões LED vão marcando se a frequência está próxima da frequência base da afinação. Quando a nota está afinada na frequência certa, o LED de cor verde fica aceso, caso contrário os LED vermelhos ficam acesos. Os LED vermelhos à esquerda e direita do LED verde mostram que a nota está abaixo ou acima da afinação correcta. O afinador também apresenta as frequências que estão a ser detectadas da guitarra, a nota correspondente e a percentagem de desvio em relação à afinação (*Percent* na Figura 61).

A janela do *Transporter* permite controlar o início e o fim de todo o processo de captação e detecção das notas. A Figura 62 mostra a janela do *Transporter*.



Figura 62: Janela Transporter para iniciar e para o processo de detecção das notas

O guitarrista prime o botão de *play* para iniciar a captação e começa a tocar a guitarra, e o botão *stop* para parar.

5.7 Testes unitários

Nesta secção trata-se a avaliação experimental da aplicação e os resultados obtidos comparativamente com os resultados esperados. Como resultado final pretende-se que um guitarrista possa tocar solos até pelo menos 180 BPM e todas as notas sejam identificadas com o mínimo de erro médio. A aplicação desenvolvida deverá produzir como resultado final o toque do guitarrista num ficheiro de imagem em formato de tablatura que representa as notas produzidas na guitarra eléctrica. O formato tablatura não possui informação de duração das notas, somente apresenta por cada linha o número do trasto que foi pressionado no braço da guitarra eléctrica.

A aplicação foi testada à medida que se desenvolveu cada componente com testes unitários. Para efectuar a avaliação desenvolveram-se pequenos testes unitários apresentados de seguida, por cada camada, classe ou componente do projecto.

Os testes foram efectuados no Linux Fedora 14 64 bits, utilizando para captar o sinal áudio um microfone, ligado à placa de som do computador. O computador possui uma única placa de som. A Tabela 17 apresenta as condições em que são executadas os testes.

Parâmetros de teste	Valor
Frequência de amostragem	44100 Hz
Tamanho do bufer do Servidor Áudio	1024
Nº de <i>Buffers</i>	2
Número de pontos FFT	1024
Janela	Hamming

Tabela 17: Condições dos testes executados

5.7.1 Testes unitários do servidor Áudio

A Tabela 18 lista os testes executados para a camada de captação do sinal áudio. Para execução dos testes está criado um projecto que serve de testes dos componentes. O projecto AudioServeTestUnit, é utilizado para criar uma aplicação de demonstração onde são testados os vários componentes da aplicação.

Classes	Testes
CAudioServer	<ol style="list-style-type: none"> 1. Detecção das placas de som existentes no computador. 2. Detecção dos drivers suportados no sistema operativo. 3. Funcionamento da classe Buffer com suporte de mais do que um <i>buffer</i>. 4. Captação do sinal vindo da placa de som usando um microfone. 5. Escrita do sinal numa instância de AudioPort adicionada ao servidor como uma porta áudio.

Tabela 18: Lista de testes unitários executados no AudioServer

Os testes foram executados utilizando JACK [73] como servidor áudio em tempo real. Com o JACK em execução qualquer aplicação que se ligar a este servidor, terá disponível os dispositivos áudio escolhidos no JACK. O JACK, por meio de sua interface gráfica, permite definir determinados parâmetros como a frequência de amostragem, o tamanho do *buffer*, o *driver* da placa de som e a placa de som que se pretende utilizar. Estes parâmetros serão partilhados com todas as aplicações que se ligarem ao JACK.

O ALSA [74] é o *driver* principal utilizado em plataformas Linux, utilizado para comunicar com a placa de som. O *driver* ALSA está disponível em qualquer distribuição Linux. A restrição do ALSA em suportar somente uma aplicação com acesso a placa de som, não possibilitou testes com este *driver* directamente, mas dado que o JACK utiliza o ALSA não haverá influência no resultado em ser um *driver* ou outro.

A Figura 64 mostra os dispositivos áudio disponibilizados pelo JACK para escrita e leitura na execução dos testes.

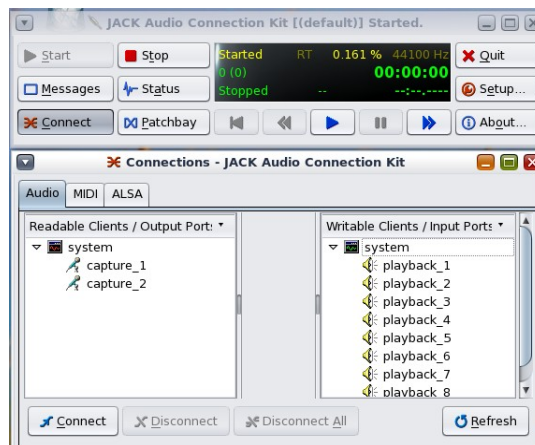
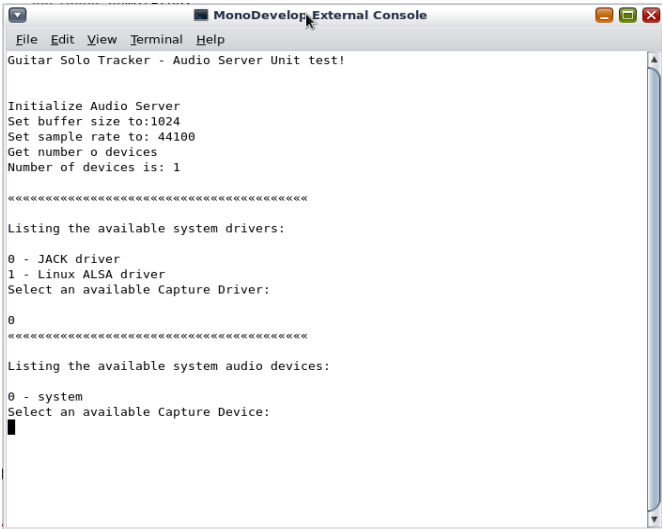


Figura 63: Interface gráfica do Jack Audio Connection Kit

A Figura 64 mostra os resultados para os testes efectuados ao AudioServer de 1 a 4, apresentados na Tabela 17. É possível verificar na Figura 65 que a aplicação lista os *drivers* áudio suportados no sistema operativo e que após escolha do JACK como *driver*, a lista de dispositivos de som mostra um único dispositivo que é *system*, que está de acordo com o que a interface do JACK apresentada na Figura 64.

Para efectuar o teste 5 da Tabela 17, que corresponde a verificar se adicionando uma porta áudio o servidor áudio escrevia correctamente, na aplicação de testes foi adicionada uma classe CDemo que deriva de CAudioPort e esta foi adicionada como sendo uma porta de

captura ao servidor áudio. Na classe `CDemo` sempre que esta recebe uma amostra, exibe o máximo valor existente no *buffer* na consola.



```
File Edit View Terminal Help
Guitar Solo Tracker - Audio Server Unit test!

Initialize Audio Server
Set buffer size to:1024
Set sample rate to: 44100
Get number o devices
Number of devices is: 1

=====

Listing the available system drivers:

0 - JACK driver
1 - Linux ALSA driver
Select an available Capture Driver:

0
=====

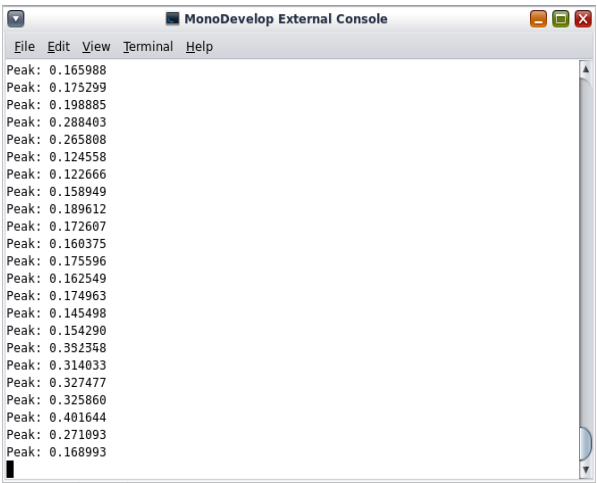
Listing the available system audio devices:

0 - system
Select an available Capture Device:
```

Figura 64: Resultado da execução do teste

Embora o JACK tenha um valor específico para o tamanho do *buffer* de captação, as portas que se ligam ao servidor áudio do *GSTTransporter* podem definir um tamanho diferente do servidor. Alguns resultados anormais foram obtidos quando o tamanho do *buffer* e da porta áudio é menor que o tamanho do *buffer* existente no servidor JACK. Foi observado que no resultado do pico máximo se obtinham valores sem significado.

A Figura 65 apresenta o resultado obtido na aplicação de teste na consola, que mostra as variações do pico máximo quando há captação de sinal.



```
File Edit View Terminal Help

Peak: 0.165988
Peak: 0.175299
Peak: 0.198885
Peak: 0.288403
Peak: 0.265808
Peak: 0.124558
Peak: 0.122666
Peak: 0.158949
Peak: 0.189612
Peak: 0.172607
Peak: 0.160375
Peak: 0.175596
Peak: 0.162549
Peak: 0.174963
Peak: 0.145498
Peak: 0.154290
Peak: 0.392358
Peak: 0.314033
Peak: 0.327477
Peak: 0.325860
Peak: 0.401644
Peak: 0.271093
Peak: 0.168993
```

Figura 65: Picos máximos do sinal captado do microfone

A lista de classes da camada de gravação do ficheiro é apresentada na Tabela 19.

Camadas/Classes	Testes
CFileIO	Gravação do sinal captado pelo servidor áudio Leitura de um ficheiro áudio

Tabela 19: Lista de testes unitários executados com CFileIO

Para efectuar o teste de gravação foi adicionada uma instância do tipo CFileIO ao projecto de testes tendo como resultado esperado um ficheiro áudio de formato WAVE contendo áudio gravado pelo microfone na execução do teste. Como resultado, o teste produziu um ficheiro *demo.wav*, cujo conteúdo tem todo o som captado no momento do teste. O teste 2 foi efectuado fazendo a leitura do ficheiro áudio gravado.

A Tabela 20 lista os testes efectuados para a classe de processamento de sinal.

Camadas/Classes	Testes
CSignalProcessing	<ol style="list-style-type: none"> 1. Cálculo de energia e pico máximo do sinal 2. Cálculo da duração do sinal 3. Estimação de frequência usando a FFTW 4. Filtro passa-baixo e decimação

Tabela 20: Lista de testes unitários executados com CSignalProcessing

Para calcular a duração da nota foi testado utilizando ficheiros com notas musicais de duração específica e observada a contabilização da duração da frequência da nota. Quando é calculada uma frequência um contador vai verificar quanto tempo se encontra a mesma frequência e com base neste valor podemos estimar a duração da nota. O valor da frequência é sempre guardado como sendo a última frequência estimada sempre a frequência última frequência é diferente da actual. Quando as frequências são iguais é incrementado o tempo de duração da mesma. O problema no cálculo era que a detecção da frequência presente para amostras do sinal da guitarra varia o valor da frequência estimada, o que leva a contagem da duração para valores diferentes dos esperados.

Para resolver este problema, foi utilizado os valores do erro máximo e mínimo obtidos da execução do *benchmark*. Assim, a comparação entre a última frequência e a frequências actual é feita tendo em consideração os valor máximo e mínimo de erro. Esta solução permitiu contabilizar a duração da frequência presente, mesmo com as pequenas oscilações no valor da frequência. O cálculo da duração não tem que ser preciso, bastando a aproximação a um valor que corresponda a configuração de batimento e quantização da música. O calculo para a duração do silêncio é feito usando o mesmo procedimento, considerando a frequência de 0 como sendo o silêncio. O silêncio é considerado quando a energia do sinal chega até um valor de limiar determinado, como por exemplo -20 dB.

Testes efectuados com amostras do sinais constituídos por dois sinais espaçados por 1 segundo de silêncio permitiu verificar se é calculada correctamente a duração dos silêncio, o resultado foi aproximado a 1 segundo.

Feitos os testes foram identificados 2 problemas: na estimação das frequências; a existência de ruído no sinal de entrada e a frequência de amostragem demasiado alta que implica baixar a resolução na frequência. Para o ruído é necessário remover estas e outras frequências que não constem na largura de banda guitarra eléctrica. Para resolver a questão da frequência de amostragem elevada foi necessário aplicar decimação ou sub-amostragem, com um filtro passa-baixo, que corresponde ao teste 4.

Depois de aplicada a decimação, os resultados da detecção de frequência melhorou, permitindo o cálculo da FFT com 512 pontos, sendo suficiente para detectar as notas musicais. Outros resultados serão apresentados no Capítulo 6, com vários parâmetros. A

execução do *benchmark*, suportada na classe *CSignalProcessing* permitiu chegar a parâmetros óptimos para a detecção de frequência.

5.7.2 Testes unitários do *Transporter*

A Tabela 21 lista os testes efectuados com o *GSTTransporter*.

Camadas/Classes	Testes
CGSTTransporter	<ol style="list-style-type: none">1. Passagem de parâmetros ao executável <i>GSTTransporter</i>2. Criação da fila de mensagem3. Execução da <i>thread</i> de processamento dos comandos4. Execução da <i>thread pool</i>5. Escrita das notas na fila de mensagem6. Início simultâneo de todas as camadas7. Escrita do ficheiro áudio8. Recepção de comandos da interface gráfica

Tabela 21: Lista de testes unitários executados com *GSTTransporter*

Para efectuar os testes do *Transporter* foi criado um projecto para testar a criação e conexão com a memória partilhada e envio mensagens de teste usando a fila de mensagens. O *Transporter* possui uma classe estática que é responsável por criar um ficheiro *log* de todo o processo que está a decorrer no executável, permitindo desta forma registar os processos que estão a decorrer dentro da aplicação. Para efectuar o primeiro teste, na entrada da aplicação é feito *log* dos parâmetros que foram passados ao *Transporter* e verificados se no *log* se os mesmos estão correctos e executam as configurações desejadas.

Para o caso dos testes 2 e 3, a criação da memória partilhada foi executada com sucesso. Para verificar se a fila de mensagem foi criada, no Linux basta ir para o directório */dev/shm* e observar se existe um ficheiro com o nome que foi atribuído à fila de mensagem. O *Transporter* possui uma classe com métodos estáticos para fazer *log* de todo o processo que decorre dentro na execução, a classe *CLogger*. Esta classe escreve num ficheiro definido, as ocorrências do *Transporter* para efeito de *debug*. Fazendo uso do *CLogger* foi possível efectuar os testes sobre os comandos enviados ao *Transporter*, via fila de mensagem, mediante a escrita no ficheiro da mensagem recebida pelo *Transporter*.

Com o programa de testes enviou-se mensagens para fila de mensagem de modo a que o *Transporter* escrevesse no ficheiro de *log* os comandos que estavam a ser recebidos. Dado que a fila de mensagens suporta somente bytes, as mensagens a serem enviadas estão limitadas a valores inteiros de 0 à 255.

Depois de criação da fila de mensagem onde serão enviadas todas as notas detectadas, a aplicação escreveu algumas notas e foi testado se seria possível ler as notas escritas pelo *Transporter* na fila de mensagem. Este teste corresponde ao número 5 e foi executado com sucesso, ou seja, foi possível depois do *Transporter* escrever uma nota musical e obter a mesma nota noutro processo usando a fila de mensagem.

O teste número 6 permite verificar se todos os componentes associados ao *Transporter* necessitam de ser iniciados simultaneamente. Como resultado pretende-se verificar que a aplicação está a escrever pelo menos o ficheiro áudio, que faz parte do teste 8.

5.7.3 Teste unitário da Interface gráfica

A Tabela 22 lista os testes efectuados para a interface gráfica.

Camadas/Classes	Testes
GuitarSoloTrackerUI	<ol style="list-style-type: none">1. Execução do executável <i>GSTTransporter</i>2. Escrita da fila de mensagem3. Recepção das notas detectadas no <i>GSTTransporter</i>4. Afinador5. Metrónomo6. Componente <i>AudioTrack</i>, verificar o desenho da onda do sinal áudio criado pelo <i>GSTTransporter</i> no componente7. Apresentação da tablatura

Tabela 22: Lista de testes unitários executados no GuitarSoloTrackerUI

O teste número 1 na Tabela 22 baseou-se em executar a aplicação. Para efectuar os 2, 3 e 4 foi necessário criar uma aplicação para aceder à memória partilhada, e dentro da interface gráfica é executada. A classe do *Transporter* é a que executa a aplicação *comm* e esta faz a escrita e leitura nas filas de mensagem.

O afinador numa primeira fase tinha dificuldades na actualização das imagens, mas funciona em pleno dependendo a afinação unicamente da precisão do *Transporter*. Os testes efectuados tinham como objectivo verificar se os vários componentes da interface gráfica eram apresentados correctamente.

O metrónomo é independente do *Transporter* e permite visualizar o batimento, com o número de batidas configurável.

O componente *AudioTrack* não possibilitava a leitura de ficheiros usando a biblioteca *libsndFile*, o que requereu a pesquisa de outra solução para leitura de ficheiros WAVE em C#. Depois de uma pesquisa foi possível encontrar uma página que contém classes em C# para leitura de ficheiros WAVE baseado na especificação, no programa WAV2GIF [75]. Com esta classe foi possível a leitura de um ficheiro WAVE e efectuar o desenho da onda, como apresenta a Figura 66.

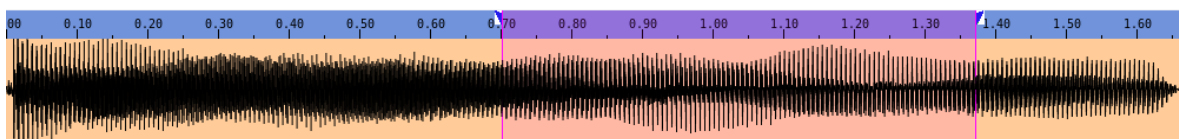


Figura 66: Screenshot do componente *AudioTrack* com desenho do sinal áudio em forma de onda

Na Figura 66 observa-se o desenho em forma de onda de um sinal áudio no componente *AudioTrack*. Este componente possui uma barra com linha do tempo por cima, para verificar. A solução usando da classe *libSndFile* não lia o conteúdo do ficheiro áudio o que pode ser um problema relacionado com a biblioteca em si, pois os testes efectuado quer com Mono quer com o .NET não funcionavam correctamente. No entanto esta solução não está descartada dado que esta biblioteca permite a leitura de variados formatos de ficheiros áudio.

Para testar a apresentação de notas na tablatura foi necessário criar um componente que desenha o formato de tablatura, porque a representação na forma de texto seria mais

complexa. Este componente, 2 na Figura 59 será actualizada sempre que receber uma nota musical, como é possível ver na Figura 59, desenha a tablatura correctamente.

6 Avaliação Experimental

Neste Capítulo apresentam-se os resultados obtidos pela aplicação. Também se analisa de forma gráfica a precisão da detecção de frequências das notas. Os testes foram efectuados considerando a captação por um microfone com uma guitarra clássica e a captação por ligação directa da guitarra eléctrica ao PC.

6.1 Execução do benchmark

Para efectuar os testes foi executado um *benchmark* da detecção de frequências, onde os parâmetros associados a classe CSignalProcessing são variados com o objectivo de se encontrar o que tem melhor resultado. A comparação dos resultados é feita utilizando valores calculados do erro médio. O erro médio é calculado somando todos os erros associados à detecção das notas. Para este efeito são produzidas frequências que representam todas as notas da guitarra eléctrica que se pretende detectar em forma de ondas sinusoidais e em seguida executa-se a detecção de cada uma das notas. O resultado da detecção é comparado com a frequência original e desta forma é calculado o erro de detecção. Estes testes também permitem ver qual é o erro mínimo e máximo que se obtém na detecção.

Foram executados *benchmark* para encontrar duas configurações diferentes: uma para detecção normal de todas as notas musicais e outra para detecção das 6 frequências correspondentes a 6 cordas da guitarra para o modo de afinação. A diferença está que no caso da afinação só pretendemos ter uma melhor precisão nas frequências correspondentes à afinação das notas, deixando as outras todas indiferentes. A Tabela 23 apresenta as frequências para afinação da guitarra, considerando uma afinação *standard*.

Notas	E	A	D	G	B	E
Frequências (Hz)	82,40	110	148	196	247	329,63

Tabela 23: Frequências de cada corda na afinação standard

Baseado na Tabela 23 verifica-se a proximidade dos valores de detecção no *benchmark* somente para as 6 frequências. Para testes com todas as frequências os valores são baseados nas frequências da Tabela 1. Os *benchmark* efectuados, foram executados variando os valores do factor de decimação e consequentemente a frequência de corte. O factor de decimação para o caso da avaliação de todas as notas da guitarra eléctrica chega a um valor máximo que respeite a frequência de Nyquist, caso contrário obtemos *aliasing* na detecção. A Tabela 24 apresenta os parâmetros utilizados para a execução do *benchmark* para detecção de todas as frequências, e as Figuras 67, 68 e 69 apresentam o gráfico de erro de estimação.

Sample Rate Hz	Buffer Size	FFT N Pontos	Factor de decimação	Down Sampling	Erro Mínimo Hz	Erro Máximo Hz	Erro Médio Hz
44100	1024	512	15	2940	-3,98	3,68	0,507
44100	1024	512	20	2205	-3.21	430,76	18,14
44100	1024	512	20	2205	-3.21	4,83	0,43

Tabela 24: Parâmetros e resultados de teste

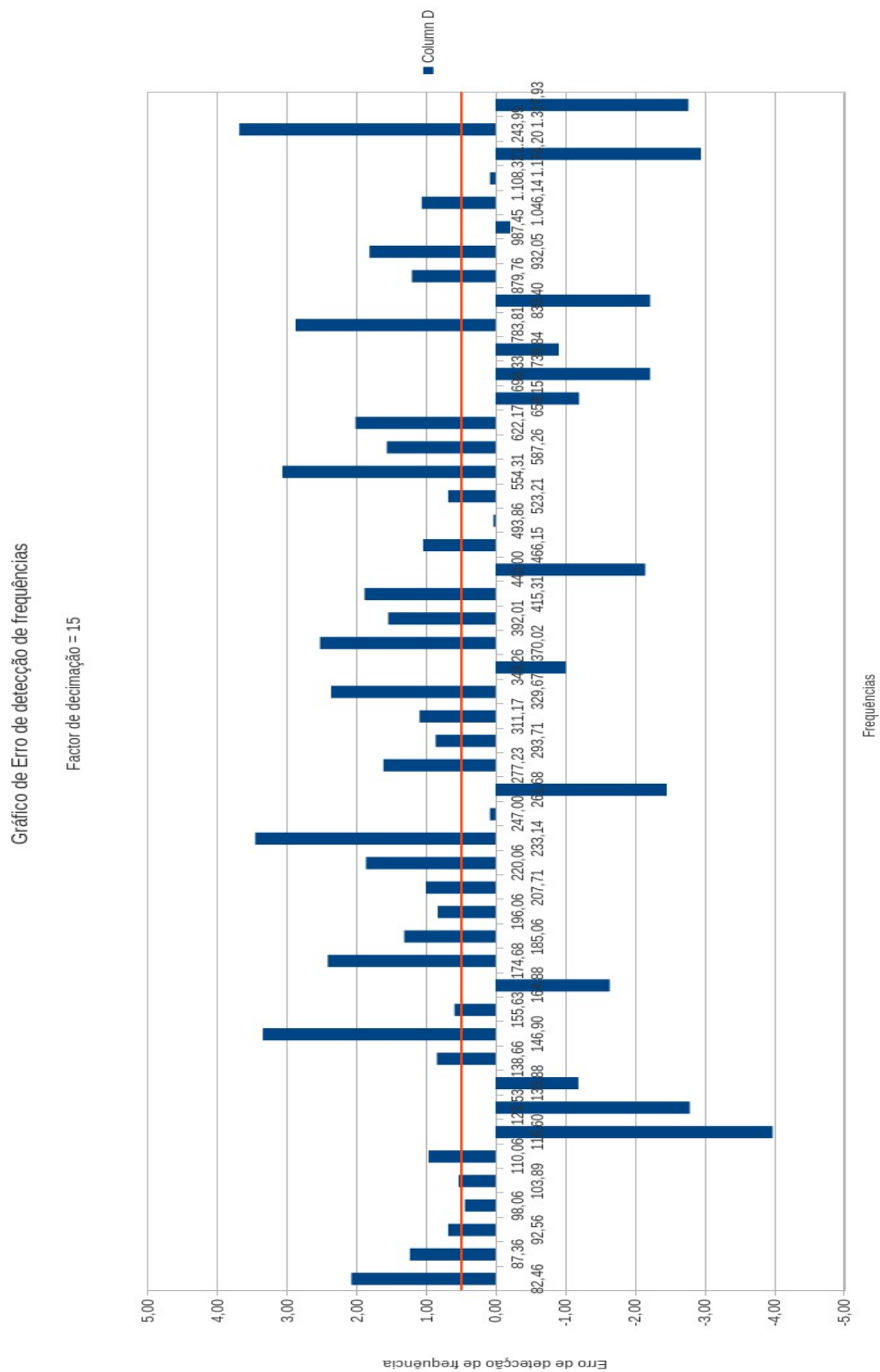


Figura 67: Erro de detecção de frequências (em Hz) com factor de decimação de 15, para as 49 notas

Na Figura 67 a linha vermelha representa o erro médio obtido no teste.



Figura 68: Erro de detecção de frequências com factor de decimação de 20, com as 49 notas

Observa-se na Figura 68 um erro máximo elevado de 430,76, resultado de *aliasing* porque a frequência de sub-amostragem não respeitar o critério de Nyquist na frequência máxima.

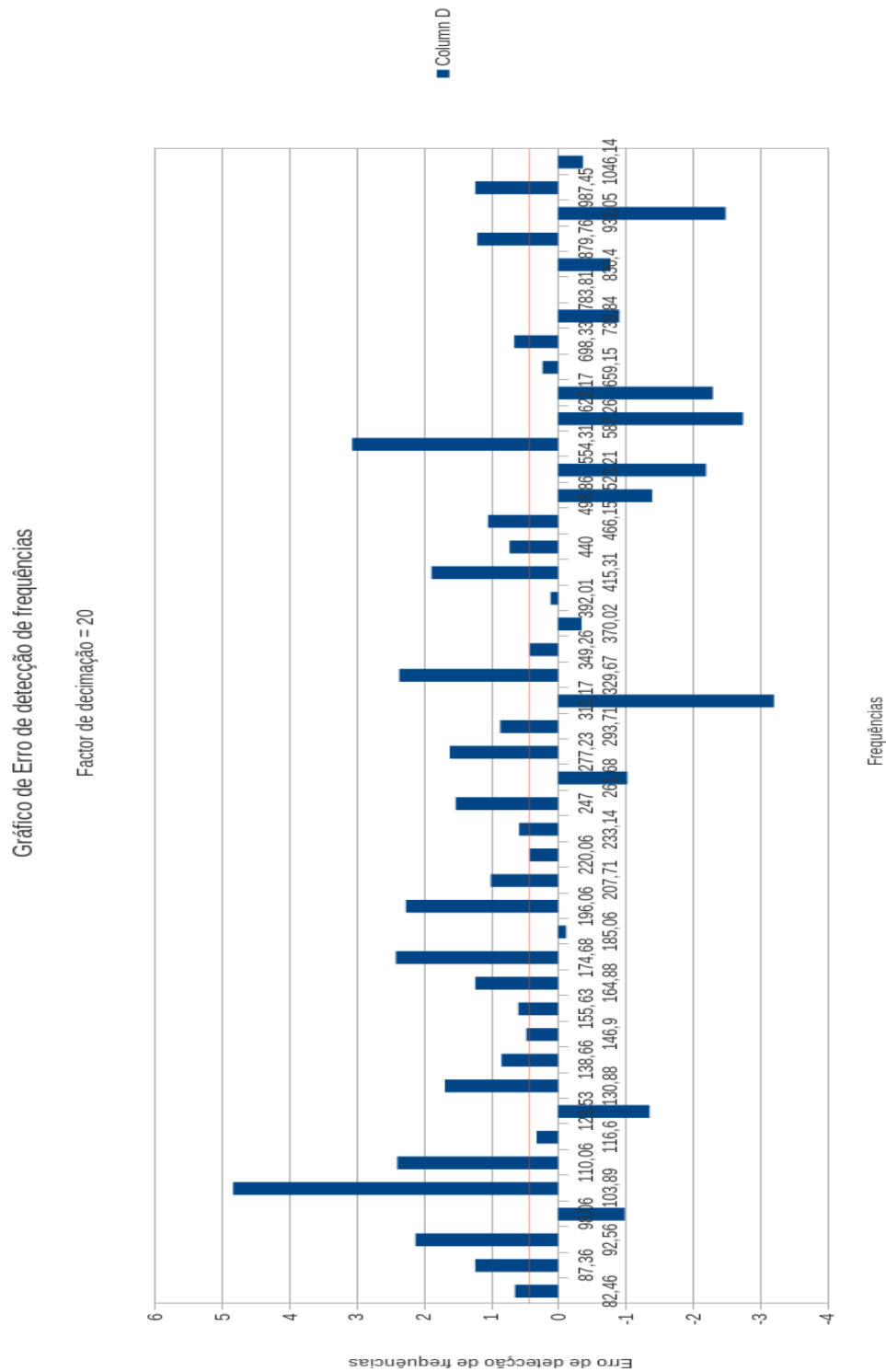


Figura 69: Erro de detecção de frequências com factor de decimação de 20, com as 45 notas descartando as notas com aliasing

Na Figura 69 temos o gráfico da Figura 68 após retirar os valores de erro muito elevado. Alguns destes erros são consequência do *aliasing* do sinal quando passa no filtro passa-baixo. Desta forma é possível analisar e comparar melhor os valores de erro. Com o gráfico

da Figura 69 podemos concluir que a partir de um factor de decimação de 20 o erro de detecção para todas as notas é elevado. No entanto, estes parâmetros podem ser úteis para a utilização na afinação da guitarra, dado que para afinar somente é necessário detectar as 6 frequências apresentadas na Tabela 23. Neste sentido, dado que no caso da afinação a frequência máxima a estimar é de 329,63 Hz, significa que é possível então aumentar o factor de decimação de modo a obter para as 6 frequências um valor de erro o mais baixo possível. A Tabela 25 resume os testes efectuados para encontrar o melhor factor de decimação.

Factor\Frequências (Hz)	82,40	110	147	196	247	329,63	Erro médio
15	2,07	0,96	3,34	0,83	0,08	2,31	1,59
16	1,72	2,4	1,56	2,28	0,61	1,31	1,64
20	0,64	2,39	0,47	2,26	1,52	2,36	1,60
25	3,22	3,26	2,19	3,12	2,38	2,36	2,75
30	2,07	3,83	3,34	3,70	0,08	2,36	2,56

Tabela 25: Resumo de resultados dos testes para encontrar o melhor factor de decimação para as 6 frequências

Na Tabela 25 o factor de decimação de 15 e 20 são os que apresentam melhor valor de erro médio absoluto. Um factor que influencia também os resultados da detecção é a precisão do tipo *float*, devido aos vários arredondamentos sofridos durante operações matemáticas nas casas decimais, mas ainda assim se consegue resultados bastante próximos da frequência que se pretende estimar. Também é necessário ter em conta que os valores de frequências produzidos pela vibração das cordas da guitarra não são exactos. Os valores de erro máximo e mínimo podem ser utilizados para melhorar a aproximação à frequência original.

Concluimos que para detecção de todas as notas, o factor de decimação de 15 apresenta melhor valor de erro máximo e mínimo, como demonstra o gráfico da Figura 67, e este é o valor utilizado pela aplicação tanto para afinação como para estimação das notas.

6.2 Afinação da guitarra

A afinação da guitarra foi testada de duas formas: captura por meio de um microfone; captura ligando a guitarra directamente ao computador. Um problema apresentado na detecção com microfone é o ruído ambiente que poderia dificultar detecção das frequências, mas o som da guitarra perto do microfone era suficiente para que o ruído ambiente não afecte em demasia os resultados. Foi necessário ajustar os valores de limiar do ruído para -25 dB para cortar o ruído ambiente e assim somente se observar alterações dos LED do afinador para os sons com potência superior à do ruído.

Para comparar se a afinação da aplicação corresponde a uma afinação próxima da *standard*, afinou-se a guitarra com o afinador Yamaha YT-120, apresentado na Figura 71, e comparou-se o resultado com o afinador da aplicação.



Figura 70: Afinador Yamaha YT-120

A Figura 71 apresenta o afinador da Yamaha YT-120 utilizado para afinar a guitarra antes de efectuar a comparação com o afinador da aplicação. É possível que diferentes afinadores tenham precisão diferente; infelizmente não existem dados técnicos referentes à precisão deste afinador. A Tabela 26 apresenta o resultado de uma guitarra afinada anteriormente com o afinador YT-120 e a comparação com o resultado obtido pelo afinador da aplicação. O objectivo é verificar de quanto o afinador da aplicação se desvia da afinação *standard* numa guitarra já afinada. Os testes foram efectuados para guitarra clássica e eléctrica.

Frequências (Hz)	82,40	110	147	196	247	329,63
Guitarra clássica Desvio (%)	2	0	1	0	2	1
Guitarra eléctrica Desvio (%)	4	1	2	2	2	4

Tabela 26: Comparação do desvio do afinador da aplicação com uma guitarra clássica e guitarra eléctrica afinadas usando o afinador Yamaha YT-120

Com a guitarra clássica, a maior dificuldade foi na detecção da frequência 196 Hz correspondente a nota G/Sol, porque na aplicação se obtinha a frequência da nota numa escala acima. Para corrigir foi necessário alterar no afinador da aplicação para aceitar também o valor 392 Hz. A frequência 82,40 Hz às vezes oscilava entre 80 a 86 tendo desvios de 1 a 4%. Para a nota ter melhor detecção era necessário tocar várias vezes a corda.

Para o caso da guitarra eléctrica, o principal problema foi a amplificação do sinal que tem grande influência na detecção da notas. Outro problema identificado na detecção das frequências 82,40 e 329,63 prende-se no facto de com a amplificação o resultado da detecção apresentava frequências de 247 Hz, que corresponde a nota B (Si). Este problema deve-se ao facto de quando a corda da guitarra vibra numa nota ela também produz outras frequências que somadas correspondem a um acorde da nota tocada; neste caso um acorde da E possui a nota B. Como a pesquisa pela nota é baseada nos máximos de potência do espectro, o que ocorre é que na componente B existe uma maior potência que a componente E. Este comportamento está directamente relacionado com o ganho de entrada da guitarra, quando aumentamos o ganho na entrada o afinador apresenta a nota E em vez de B.

Também se observou que a escrita de notas na fila de mensagem e o facto destas quando estão cheias não permitirem a escrita até que sejam lidas as mensagens, para o caso específico do afinador não permite descartar as *frames* antes pelas *frames* actuais, o que torna a actualização da interface gráfica não correspondente com o que foi tocado na guitarra.

Conclui-se que a afinação da aplicação consegue uma boa precisão, mas que sofre grande influência do ganho do sinal. As notas foram detectadas tanto com guitarra clássica como com guitarra eléctrica, onde esta última apresenta mais dificuldade na detecção quando ligada ao computador sem amplificação externa. Constatou-se também que é possível detectar as notas de acordes sem que fosse necessário determinar as notas que compõe o acorde, ou seja, dado que um acorde é constituído por várias notas, o somatório destas determinam a nota base do acorde; desta forma quando é tocada um acorde, a nota base que dá o nome ao acorde é detectada. Recomenda-se para o caso da guitarra eléctrica que a afinação da mesma seja feita sem efeito, ou seja, com som original da guitarra para que o resultado seja fiável.

6.3 Detecção e apresentação das notas

Nesta secção analisa-se a detecção e apresentação das notas musicais enquanto a guitarra é tocada em real-time. Para efeito os testes são efectuados com guitarra eléctrica e guitarra clássica. O objectivo é obter o desenho da tablatura enquanto a guitarra é tocada. A tablatura pode ser gravada em formato XML e também como uma imagem. Para efeito de testes são utilizados as configurações apresentadas na Tabela 24.

6.3.1 Testes efectuados com a guitarra clássica

O primeiro teste efectuado com a guitarra clássica, com a captação por meio de um microfone mostrava nas posições corrente do cursor valores dos trastos mas de forma pouco perceptível para que se pudesse realmente estabelecer uma comparação de aproximação ao que foi tocado na guitarra clássica, como apresenta a Figura 71, notas tocadas com 40 BPM.

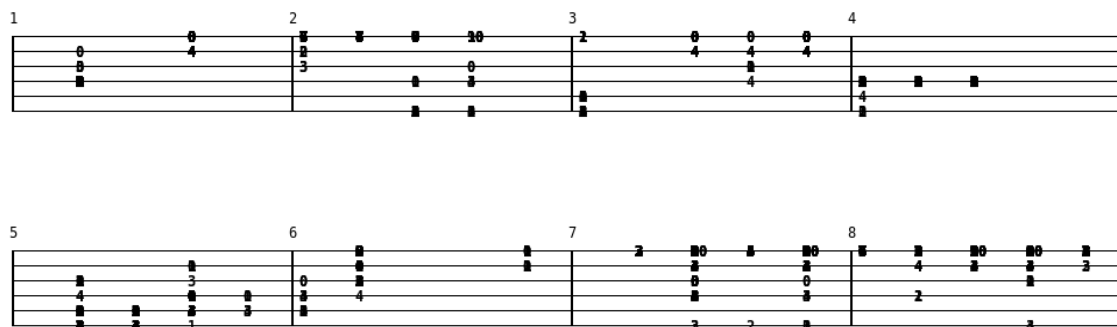


Figura 71: Desenho da tablatura como resultado de notas tocadas na guitarra clássica

Podemos observar na Figura 71 que por várias vezes as notas estão sobrepostas o que retira a visibilidade do resultado. Este problema decorre pelo facto de a todo momento se está a determinar 6 frequências independentemente de na guitarra só ter sido tocada uma nota. O que ocorre muitas vezes, é que restantes frequências encontradas ficam sobrepostas na mesma corda e posição, como é visível na Figura 71. A solução para este problema implica a necessidade de melhoria na pesquisa de mais do que um máximo no espectro de potência da FFT no *Transporter* e também eliminação de ruído.

Depois de alterado a aplicação para somente mostrar uma frequência, o teste foi executado tocando cada corda da guitarra sem trasto premidos. Os resultados foram inconclusivos dado que não foi possível verificar se a nota apresentada correspondia a nota correcta, dado a quantidade de notas apresentadas na tablatura não estarem sincronizadas com a nota produzida na guitarra. Também não foi possível encontrar um padrão que permitisse identificar um comportamento específico quando uma nota específica é tocada na guitarra.

6.3.2 Testes efectuados com guitarra eléctrica

Os testes efectuados com a guitarra eléctrica não foram também conclusivos pelo mesmos factores da guitarra clássica. Durante os testes deparou-se com o problema das frequências estimadas estarem algumas vezes uma ou das oitavas acima. A vibração das cordas faz variar a nota por várias frequências o que dificulta a escolha da nota tocada. A amplificação do sinal da guitarra eléctrica também é importante porque tem grande influência no resultado da estimação das frequências. O ganho pode ser controlado pelo volume do microfone disponibilizado pelo sistema operativo.

Outro problema encontrado está relacionado com BPM configurado para desenhar a tablatura. Se o valor de BPM for baixo em relação ao que o guitarrista está a tocar, ocorre a sobreposição de notas enquanto o cursor da tablatura está na mesma posição. Isso significa que o guitarrista que o número de batimentos configurado tem que ser igual ou maior do que o guitarrista está a tocar, ou aumentar o valor da quantização. Este problema também está relacionado com a sincronia entre a escrita de notas na fila de mensagem e a leitura do *Transporter*.

Em relação a fila de mensagem de *frames*, o facto de não se aceder directamente a fila de mensagem e ter que se usar um processo para fazer a leitura da mesma num período específico de tempo torna a escrita das notas pouco precisas, porque o momento que é feito a leitura da *frame* pode não coincidir com o tempo correcto na tablatura. Foi possível observar no ficheiro *log* do *Transporter* que muitas *frames* não são enviadas para fila de mensagem por esta estar cheia, o que implica que a taxa de escrita da fila de mensagem e de leitura não são proporcionais. Uma possível solução para este problema foi aumentar o tamanho da fila de mensagem do valor inicial de 1000 para 2000 e aumentar também o ritmo de leitura da fila de mensagem por meio do aumento do BPM para 100. No entanto, o aumento do tamanho da fila de mensagem aumenta também a latência de actualização da nota na interface gráfica. Resumindo, para o problema de sincronismo ser resolvido é necessário implementar o acesso directo a fila de mensagem em C#.

7 Conclusões

Este trabalho tinha como objectivo desenvolver uma aplicação que fosse capaz de captar o sinal da guitarra eléctrica e apresentar ao utilizador as notas tocadas pelo guitarrista. O objectivo da aplicação é ajudar o guitarrista a lembrar as notas tocadas numa música num momento de inspiração ou improviso. Sem esta aplicação, o guitarrista teria que gravar e ouvir várias vezes o que tocou para registar as notas tocadas.

Dadas as características da guitarra eléctrica, principalmente do sinal produzido pela mesma a solução para detectar as notas tocadas pelo guitarrista passou por determinar as frequências geradas pelas cordas da guitarra quando tocadas. Para resolver este problema, foram analisados 3 algoritmos de estimação de frequência: Goertzel, auto-correlação e a FFT. Da análise dos 3 algoritmos estudados, os resultados levaram a escolher a FFT para estimação de frequência. Para o desenvolvimento do projecto foi utilizada a biblioteca FFTW para cálculo da FFT.

Todo o processo de detecção de frequências, foi elaborado tendo em conta as técnicas aprendidas durante disciplinas de processamento áudio e de programação. Estas técnicas incluem *double buffering*, janela, filtragem, *multithreading* e comunicação entre processos. A aplicação foi desenvolvida usando as linguagens, C/C++ e C#; C++ foi utilizada para programação de troços de código críticos relacionados com o processamento de sinal, e C# para o desenvolvimento da interface gráfica.

Neste desenvolvimento não se tirou partido ainda da arquitectura *multicore* por meio de programação paralela de forma explícita, no entanto foram utilizadas técnicas de programação concorrente. A aplicação de programação paralela será estudada num trabalho futuro para melhorar o desempenho da aplicação em computadores com multi-processador ou *multicore*, caso num trabalho futuro se revelar necessário.

A abordagem da solução foi baseada numa arquitectura servidor/cliente. O servidor é uma aplicação que se executa num processo independente e o cliente corresponde à interface gráfica que também se executa num processo independente. A principal vantagem é o desacoplamento entre as duas partes, principalmente pelo facto de estarem desenvolvidas em linguagens diferentes. Desta forma também permite que outra aplicação possa ligar ao servidor e receber as notas musicais detectadas.

O processo mais crítico foi a precisão na detecção das notas e estimação da sua duração das notas. O processamento do sinal gerado pela guitarra foi efectuado em modo *online*, quando o guitarrista está a tocar. Os testes revelaram que para afinação era possível estimar as frequências correspondentes à afinação *standard* com uma percentagem de erro máximo de 2 e 4 Hz para guitarra clássica e guitarra eléctrica respectivamente. No entanto para a criação da tablatura, o principal problema estava no facto de não se conseguir distinguir a nota tocada com qualquer padrão no resultados apresentados. A existência de uma diferença no intervalo de leitura por parte da interface gráfica faz com várias notas seriam perdidas, levando a necessidade de leitura das notas de uma forma mais directa e mais síncrona com o processamento de sinal.

Existem melhorias a serem implementadas para melhor apresentar a tablatura, que fará parte do trabalho futuro a ser desenvolvido. Numa primeira fase será resolver a questão da fila de mensagem de modo a permitir menor perdas notas por parte do escritor por causa da demora do leitor. Para este efeito será implementada a fila de mensagem com base na biblioteca Boost Interprocess::message_queue em C# usando mapeamento de memória em ficheiro.

Terá que ser melhorado o *Transporter* de modo a que esteja sincronizado com o batimentos escolhidos pelo guitarrista, enviar *frames* somente quando o sinal for considerando uma nota, baseando num valor limiar para determinar se o sinal é ou não uma nota. Neste sentido, sempre que a *frame* captada não for considerada uma nota não é enviada para a fila de mensagem. Na aplicação não é utilizado dados referentes a duração da nota, mas no trabalho futuro estes dados serão usados para implementação da partitura.

O trabalho futuro irá envolver a detecção de algumas técnicas que o guitarrista pode executar durante o improviso, tais como *bending*, *slide*, *tremolo*. Para efectuar a detecção destas técnicas é necessário fazer uma análise do sinal da guitarra, estudar como o sinal varia para cada uma das técnicas, e com base neste estudo analisar técnicas que permitam detectar variações do sinal por forma a identificar a técnica em causa.

O trabalho futuro também irá investigar algoritmos ou soluções para optimização das notas na tablatura, tendo em conta a estrutura das mãos e o posicionamento das notas no braço da guitarra. Este estudo tem como objectivo diminuir o número de notas que têm que ser trasladadas para melhor optimização quando o guitarrista estiver a tocar a guitarra.

Este problema existe porque no braço da guitarra, a partir do 5º trasto de cada corda todas as notas podem ser encontradas nas cordas seguintes, na orientação das cordas de baixo para cima, onde a corda mais baixa (E) representa a nota (E) mais grave na guitarra, como apresenta a Figura 72 [76].



Figura 72: Braço da guitarra com as notas correspondentes a cada corda nos trastos (adaptado de [76])

Observa-se na Figura 72 que a partir do 5º trasto onde tem a nota A todas as notas seguintes aparecem na corda seguinte. Neste caso a nota A na corda E corresponde a tocar a corda A aberta, e C na corda E, tocada no 8º trasto corresponde a C tocada na corda A no terceiro trasto. Significa que as notas de uma corda num dado trasto, correspondem à mesma nota noutra trasto adiante ou atrás. Neste caso se a aplicação detectar a nota C no 8º trasto da corda E pode também escrever como C no 3º trasto da corda A. Neste caso se for detectada a nota C da corda E no oitavo trasto, e a nota seguinte for a nota A no 2º trasto da corda G, é mais prático para o guitarrista tocar C no 3º trasto da corda A, por causa da distância entre os trastos.

8 Anexo A - Tabelas com resultados de detecção de frequência

Frequência Original	Frequência estimada	Índices	Erro
82,46	81,83	14	2,07
87,36	86,13	15	1,23
92,56	90,44	16	0,68
98,06	99,05	17	0,44
103,89	99,05	18	0,53
110,06	107,67	19	0,96
116,6	116,28	21	-3,98
123,53	124,89	22	-2,79
130,88	129,2	23	-1,19
138,66	137,81	24	0,84
146,9	146,43	25	3,34
155,63	155,04	27	0,59
164,88	163,65	29	-1,64
174,68	172,27	30	2,41
185,06	185,19	32	1,31
196,06	193,8	34	0,83
207,71	206,72	36	1,00
220,06	219,64	38	1,86
233,14	232,56	40	3,45
247	245,48	43	0,08
261,68	262,71	46	-2,46
277,23	275,63	48	1,61
293,71	292,85	51	0,86
311,17	314,38	54	1,09
329,67	327,3	57	2,36
349,26	348,84	61	-1,01
370,02	370,37	64	2,52
392,01	391,9	68	1,54
415,31	413,44	72	1,88
440	439,28	77	-2,15
466,15	465,12	81	1,04
493,86	495,26	86	0,03
523,21	525,41	91	0,68
554,31	551,25	96	3,06
587,26	590,01	102	1,56
622,17	624,46	108	2,01
659,15	658,92	115	-1,20

Tabela 27: Resultado de detecção de frequências com factor de decimação de 15. As frequências em amarelo correspondem às de afinação standard

Frequência Original	Frequência estimada	Índices	Erro
82,46	81,83	19	0,64
87,36	86,13	20	1,23
92,56	90,44	21	2,12
98,06	99,05	23	-0,99
103,89	99,05	23	4,83
110,06	107,67	25	2,39
116,6	116,28	27	0,32
123,53	124,89	29	-1,36
130,88	129,2	30	1,68
138,66	137,81	32	0,84
146,9	146,43	34	0,47
155,63	155,04	36	0,59
164,88	163,65	38	1,23
174,68	172,27	40	2,41
185,06	185,19	43	-0,12
196,06	193,8	45	2,26
207,71	206,72	48	1
220,06	219,64	51	0,42
233,14	232,56	54	0,58
247	245,48	57	1,52
261,68	262,71	61	-1,03
277,23	275,63	64	1,61
293,71	292,85	68	0,86
311,17	314,38	73	-3,21
329,67	327,3	76	2,36
349,26	348,84	81	0,42
370,02	370,37	86	-0,35
392,01	391,9	91	0,11
415,31	413,44	96	1,88
440	439,28	102	0,72
466,15	465,12	108	1,04
493,86	495,26	115	-1,4
523,21	525,41	122	-2,2
554,31	551,25	128	3,06
587,26	590,01	137	-2,75
622,17	624,46	145	-2,3
659,15	658,92	153	0,23
698,33	697,68	162	0,65
739,84	740,74	172	-0,91
783,81	783,81	182	0
830,4	831,18	193	-0,78
879,76	878,55	204	1,2
932,05	934,54	217	-2,49
987,45	986,22	229	1,23
1046,14	1046,51	243	-0,37
1108,32	1098,19	255	10,13
1174,2	1029,29	239	144,91
1243,99	960,38	223	283,61
1317,93	887,17	206	430,76

Tabela 28: Resultado de detecção de frequências com factor de decimação de 20. As frequências em amarelo correspondem às de afinação standard

9 Anexo B - Cálculos dos coeficientes

Para calcular os coeficientes do filtro de Chebyshev do tipo I o ganho da resposta em frequência é dada por

$$G_N(\omega) = |H_N(j\omega)| = \frac{1}{\sqrt{1 + \varepsilon^2 T_N^2(\omega)}}, \quad (28)$$

onde N é a ordem do filtro, ω a frequência angular de corte, ε é o factor de *ripple* da banda de passagem, T_n representa o polinómio de Chebyshev de ordem N e é definido por

$$T_N(x) = \begin{cases} \cos(N \arccos x), & |x| \leq 1, \\ \cosh(N \operatorname{arccosh} x), & |x| > 1, \end{cases} \quad (29)$$

e satisfaz a recursividade

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{N+1}(x) &= 2xT_N(x) - T_{N-1}(x). \end{aligned} \quad (30)$$

Para $N \geq 1$, satisfaz as seguintes relações:

1. Para $|x| \leq 1$, o ganho do polinómio variam entre +1 e -1.
2. Para $|x| > 1$, o ganho incrementa monotonicamente com x .
3. $T_N(1) = 1$ para todo o N .
4. $T_N(0) = \pm 1$ para N par e $T_N(0) = 0$ para N ímpar.
5. $T_N(x)$ Cruza o zero no intervalo de $-1 \leq x \leq 1$.

Para determinarmos a localização dos pólos calcula-se ε factor de *ripple* com base no valor escolhido em dB na banda de passagem.

$$\varepsilon = \sqrt{10^{\text{ripple em db}/10} - 1}. \quad (31)$$

Partindo da propriedade 1 do polinómio de Chebyshev sabemos que para os pólos ocorrem quando

$$1 + \varepsilon^2 T_N^2(\omega) = 0 \rightarrow T_N = \pm \frac{j}{\varepsilon}. \quad (32)$$

Fazendo $\omega = \cos(\phi)$, significa que $\phi = \arccos(\omega)$, e no domínio dos números complexos obtemos

$$\phi = \arccos(\omega) = u + jv \quad (33)$$

Relacionando (23) em 22, obtemos

$$T_N = \cos(N\phi) = \cos(Nu) \cosh(Nv) - j \sin(Nu) \sinh(Nv) = \pm \frac{j}{\varepsilon}, \quad (34)$$

o que significa que a parte real de T_N em (24) tem que ser igual a zero. Então é necessário que

$$\cos(Nu)\cosh(Nv)=0 \quad (35)$$

o que implica que $\cos(Nu)=0$. Para satisfazer esta condição significa que u terá os seguintes valores

$$u=u_k=(2k+1)\pi/2N, k=0,1,\dots,N-1. \quad (36)$$

Para estes valores de u , $\sin(Nu)=\pm 1$, então temos

$$\sinh(Nv)=\frac{1}{\varepsilon}, \quad (37)$$

que implica que v tem o seguinte valor

$$v=v_0=\frac{\operatorname{arsinh}(\frac{1}{\varepsilon})}{N}. \quad (38)$$

Definindo $s=j\omega$ obtemos

$$s=j\omega=j\cos(\phi)=j\cos(u+jv)=j\cos((2k+1)\frac{\pi}{2N}+jv_0) \quad (39)$$

que dá a localização dos N pólos no plano complexo como

$$s_k=\sigma_k+j\omega_k \quad (40)$$

onde

$$\sigma_k=-\sinh(v_0)\cos(\frac{k\pi}{2N}) \quad (41)$$

e

$$\omega_k=\cosh(v_0)\sin(\frac{k\pi}{2N}) \quad (42)$$

para N valores de k onde

$$k=\pm 1, \pm 3, \pm 5, \dots, \pm(N-1) \quad N \text{ par} \quad (43)$$

e

$$k=\pm 0, \pm 2, \pm 4, \dots, \pm(N-1) \quad N \text{ ímpar}. \quad (44)$$

A resposta em frequência para o filtro é dada por

$$F(s)=\prod_k \frac{1}{s^2-2\sigma_k s+(\sigma_k^2+\omega_k^2)}. \quad (45)$$

Referências

- 1: Leggato, Técnica de legato na guitarra,03, 2011, <http://www.5min.com/Video/Understanding-the-Legato-Technique-68806222>
- 2: Youtube, Técnica de shredding na guitarra,06, 2010, <http://www.youtube.com/watch?v=blhz99V6e94>
- 3: Guitar Pro, Guitar Tablature editor, Scale Tools, chord diagrams, MIDI and ASCII import/export,03, 2011, <http://www.guitar-pro.com/en/index.php>
- 4: MIDI, MIDI,03, 2011, <http://www.midi.org/aboutmidi/index.php>
- 5: Guitar Rig, Software para efeitos de guitarra em tempo real no PC.,03, 2011, <http://www.native-instruments.com/#/en/products/guitar/guitar-rig-4-pro/>
- 6: Roland VG-99, Guitar synth from Roland,06, 2010, <http://www.roland.com/products/en/VG-99/features.html>
- 7: Roland, Roland RG-99,09, 2010, <http://www.roland.com/products/en/VG-99/index.html>
- 8: Roland, Roland PickUps,09, 2010, <http://www.roland.com/products/en/GK-3/index.html>
- 9: MonoDevelop, MonoDevelop IDE,02, 2011, <http://monodevelop.com/>
- 10: S. Kuo, B. Lee, Real-Time Digital Signal Processing, 2001, Wiley
- 11: Guitar Master Class, Braço guitarra,09, 2010, http://www.guitarmasterclass.net/guitar_forum/uploads/post-1167-1192068497.jpg
- 12: Eurico A. Cebolo, Livro Solfejo Mágico, 2002, Eurico A. Cebolo
- 13: Metrônomo, Metrônomo analógico,09, 2010, <http://umcantinhomviolao.files.wordpress.com/2009/06/metronomo801ty9.jpg>
- 14: KORG, Metronome Digital TM-40,01, 2010, <http://korg.com/product.aspx?&pd=209>
- 15: Pure Lessons, Escala de piano,02, 2011, http://www.purelessons.com/theory_sections.php
- 16: Figuras musicais, ,09, 2010, <http://batera.files.wordpress.com/2006/10/figuras-e-pausas-musicais.gif?w=450>
- 17: Adnipo Isesaki, Exemplo de compassos,01, 2011, <http://www.adnipo-isesaki.com/?secao=aulas-demusica&id=15>
- 18: Relação figuras, Relação entre figuras musicais,09, 2010, http://4.bp.blogspot.com/_qWI-LEZ71m7k/RqnhjCrcgII/AAAAAAAAAEs/B2VIgbkTHVY/s320/dicas_teorias3aa.GIF
- 19: Slide, Técnica slide na guitarra,04, 2010, <http://www.5min.com/Video/Lessons-For-Electric-Slide-Guitar-In-Standard-Tuning-162417200>
- 20: Bending, Técnica de bending na guitarra,02, 2010, <http://www.youtube.com/watch?v=DiqA-pItNtzg>
- 21: Tremolo, Efeito tremolo na guitarra.,06, 2010, http://www.ehow.co.uk/video_2378434_tremolo-effects-electric-guitar.html
- 22: Dario Cortese, Guitar Lesson,01, 2011, <http://www.dariocortese.com/lessons/guitarlessons/licks/2010-2/september/>
- 23: www.howtoreadguitartabs.net, Tab Legend,01, 2011, <http://www.howtoreadguitartabs.net/guitar-tab-legend.html>
- 24: Cyfuss, Guitarra Eléctrica,03, 2011, http://www.cyfuss.com/curso_teorico_practico_de_musica/partes_de_la_guitarra_electrica
- 25: Howstuffworks, Funcionamento dos captadores,09, 2010, <http://entertainment.howstuffworks.com/electric-guitar1.htm>

- 26: Stellfner, Esquema de um captador,01, 2011, <http://www.stellfner.com.br/Conceitos%20Pickup%20Teoria.htm#magneto>
- 27: Oppenheim, A., Schafer, R., Discrete-Time Signal Processing, 1999,Prentice Hall
- 28: Connexions, Autocorrelação,02, 2011, <http://cnx.org/content/m10676/latest/>
- 29: Patrick F. Dunn, Measurement and Data Analysis for Engineering and Science, 2005,McGraw–Hill
- 30: Tony Fisher, Interactive Digital Filter Design,01, 2011, <http://www-users.cs.york.ac.uk/~fisher/>
- 31: Tony Fisher, Butterworth / Bessel / Chebyshev Filters,01, 2011, <http://www-users.cs.york.ac.uk/~fisher/mkfilter/trad.html>
- 32: Giovanni Bianchi and Roberto Sorrentino, Electronic filter simulation & design, 2007,McGraw-Hill Professional
- 33: Steven W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing, 1997,DSP Guide
- 34: Enotes, Chebyshev filter,02, 2011, http://www.enotes.com/topic/Chebyshev_filter
- 35: Connexions, Chebyshev Filter Properties,02, 2011, <http://cnx.org/content/m16906/1.1/>
- 36: Wikipedia, Diagrama Forma Directa II,02, 2011, http://en.wikipedia.org/wiki/Digital_filter
- 37: Julius O. Smith III, Análise Espectral,02, 2011, https://ccrma.stanford.edu/~jos/sasp/Spectrum_Analysis_Windows.html
- 38: PThread-Win32, POSIX 1003.1-2001 Pthread implementation for Windows.,01, 2010, <http://sourceware.org/pthreads-win32/>
- 39: Boost C++ Library, Conjunto de bibliotecas C++,06, 2010, <http://www.boost.org/>
- 40: Boost Thread, Biblioteca para programação concorrente.,04, 2010, http://www.boost.org/doc/libs/1_43_0/doc/html/thread.html
- 41: Boost Signals, Biblioteca para implementação de notificação publisher/subscriber.,04, 2010, http://www.boost.org/doc/libs/1_43_0/doc/html/signals.html
- 42: Boost Signals2, Variante thread-safe da biblioteca Boost Signals.,04, 2010, http://www.boost.org/doc/libs/1_43_0/doc/html/signals2.html
- 43: OpenMP, Biblioteca em C para programação paralela, em sistemas multicore ou multiprocessadores, com memória compartilhada.,01, 2009, <http://openmp.org>
- 44: Prof. Alan Kaminsky Department of Computer ScienceRochester Institute of Technology , Multi-processador,09, 2010, <http://www.cs.rit.edu/~ark/lectures/pj03/notes.shtml>
- 45: FFTW, Biblioteca em C/C++ que implementa o cálculo da FFT (DFT) em uma ou mais dimensões.,5, 2009, <http://www.fftw.org/>
- 46: SIMD, ,09, 2010, <http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingTutorial/BasicsOfSIMDProgramming.html>
- 47: RTAudio, C++ classes that provide a common API (Application Programming Interface) for real-time audio input/output across Linux, Macintosh OS-X and Windows .,06, 2010, <http://www.music.mcgill.ca/~gary/rtaudio/>
- 48: Libsndfile, Biblioteca Libsndfile,09, 2010, <http://www.mega-nerd.com/libsndfile/>
- 49: boost program_option, boost program_option,09, 2010, http://www.boost.org/doc/libs/1_44_0/doc/html/program_options.html
- 50: Boost threadpool, Boost Threadpool,09, 2010, <http://threadpool.sourceforge.net/>
- 51: Boost Interprocess, Boost Interprocess,04, 2010, http://live.boost.org/doc/libs/1_44_0/doc/html/interprocess.html
- 52: W3, XML,03, 2011, <http://www.w3.org/XML/>

53: Boost Timer, Biblioteca Boost para funções de clock.,06, 2010, http://www.boost.org/doc/libs/1_43_0/libs/timer/timer.htm

54: timeGetTime, Função da API do Windows para retornar o relógio do sistema operativo.,05, 2010, <http://msdn.microsoft.com/en-us/library/dd757629%28VS.85%29.aspx>

55: GetTickCount, Função da API do Windows para retornar o número em milissegundos desde de que o sistema operativo iniciou.,05, 2010, <http://msdn.microsoft.com/en-us/library/ms724408%28VS.85%29.aspx>

56: QueryPerformanceCounter, Retorna o valor actual do contador de alta resolução.,05, 2010, <http://msdn.microsoft.com/en-us/library/ms644904%28VS.85%29.aspx>

57: clock_gettime, retorna o conjunto do tempo de um clk_id específico.,05, 2010, http://linux.die.net/man/3/clock_gettime

58: FFTW , FFTW 3.3,03, 2011, <http://www.fftw.org/doc/Introduction.html>

59: .Net Interop, .Net Interop,09, 2010, <http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.aspx>

60: DllImport, DllImport,09, 2010, <http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.aspx>

61: boost file_lock, Boost File Lock,09, 2010, http://www.boost.org/doc/libs/1_43_0/doc/html/interprocess/synchronization_mechanisms.html#interprocess.synchronization_mechanisms.file_lock

62: MicroSoft, .Net Memory Mapped File,02, 2011, <http://msdn.microsoft.com/en-us/library/system.io.memorymappedfiles.memorymappedfile.aspx>

63: GTK sharp, GTK-Sharp,09, 2010, <http://www.mono-project.com/GtkSharp>

64: LilyPond, LilyPond music notation libray,02, 2011, <http://lilypond.org/>

65: Music XML, MusicXML,02, 2011, <http://www.musicxml.org/>

66: ABC Notation, ABC Notation,02, 2011, <http://abcnotation.com/>

67: VexFlow, VexFlow Music Engraving in JavaScript and HTML5,02, 2011, <http://vexflow.com/>

68: Microsoft, ProcessStartInfo,02, 2010, http://msdn.microsoft.com/en-us/library/system.diagnostics.processstartinfo_properties.aspx

69: Microsoft, DllImport,02, 2011, <http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.aspx>

70: Microsoft, StructLayout,02, 2011, <http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.structlayoutattribute.aspx>

71: Microsoft, IntPtr,02, 2011, <http://msdn.microsoft.com/en-us/library/system.intptr%28v=VS.100%29.aspx>

72: Microsoft, MarshalAs,02, 2011, <http://msdn.microsoft.com/en-us/library/s9ts558h%28v=VS.100%29.aspx>

73: Jack Audio Connection Kit, Jack Audio Connection Kit,06, 2010, <http://jackaudio.org/>

74: ALSA (Advance Linux Sound Architecture), Arquitectura áudio e MIDI para Linux.,06, 2010, <http://www.alsa-project.org/>

75: Stuart Cam, Wav2Gif,02, 2011, <http://blog.codebrain.co.uk/post/2009/04/30/C-Sharp-WAV2-GIF.aspx>

76: Guitar Fretboard Diagram, Guitar Fretboard Diagram,02, 2011, <http://www.guitarfretboarddiagram.com/>